Space-Time Algorithms: Semantics and Methodology

Marina Chien-mei Chen

Department of Computer Science
California Institute of Technology

5090-TR-83

Space-Time Algorithms: Semantics and Methodology

Thesis by

Marina Chien-mei Chen

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

5090:TR:83

California Institute of Technology

Pasadena, California

1983

(Submitted May 5, 1983)

To My Grandmothers and Parents

Acknowledgments

I wish to thank Professor Carver Mead, my advisor, for being a great teacher and a good friend, for his guidance, encouragement, and his many insights which made this work possible. He has spent enormous amount of energy on developing my proficiency in scientific research and enthusiasm in teaching. His vision and philosophy will always be a part of me.

I wish to thank my office-mate Young-il Choo for his continuous help during my graduate studies. I have benefited from many of his ideas and from my discussions with him. The experience of teaching the "Formal models of computation" course in 1981 together with him has been a most significant experience in my personal growth.

I also wish to thank other members of the faculty of Computer Science Department: Alain Martin for his valuable comments and his patience in helping with the presentation of the thesis; Lennart Johnsson for his encouragement and constant interest in this work; Randy Bryant for his insight in discussions on the subject of simulation; Jim Kajiya for his encyclopedic knowledge of the literature; Chuck Seitz for introducing me to his work on self-timed systems; and Fred Thompson for having taught me about language processing and exposed me to natural language systems. Finally, my thanks goes to Ronald Graham for many of his inspiring ideas.

This work is sponsored by the System Development Foundation, an IBM Doctoral Fellowship from January 1982 to September 1982, and its initial phase is sponsored by the Defense Advanced Research Projects Agency ARPA Order #3771 and monitored by Office of Naval Research Contract #N00014-79-C-0597.

Abstract

A methodology for specifying concurrent systems is presented. A model of computation for concurrent systems is presented first. The syntax and semantics of the language CRYSTAL are introduced. The specification of a system is called a space-time algorithm since space and time are explicit parameters in the description. Fixed-point semantics is used for abstracting the behavior of a system from its implementation. The consistency between an implementation and its description can therefore be ensured using this method. Formal semantics for an arbitrary transistor network is given. An "interpreter" for space-time algorithms — a hierarchical simulator — for VLSI systems is presented. The framework can be viewed as a concurrent programming notation when describing communicating processes and as a hardware description notation when specifying integrated circuits.

Contents

Chapter 1

Introduction

Recent developments in the technology of fabricating large-scale integrated circuits have made it possible to implement computing systems that use many hundred thousand transistors to achieve a given task. An interesting design will have high computational complexity rather than merely a vast number of identical simple components such as memory elements. Such a design can be represented as a fully instantiated implementation of objects of the implementation medium (e.g. transistors in VSLI technology) or as successive hierarchical levels of implementations where each level is constructed of objects which are abstract models of the implementation at the level below it. The former allows implementation details at the bottom level to penetrate throughout the whole design. Such a representation may be suited for machine execution but is hard to deal with from a designer's point of view, and verifying both its functionality and physical layout is costly. As the complexity of the design grows, the limitation of this approach becomes more apparent. The second approach is aimed at managing the complexity of a design. The design is partitioned into successive levels of sub-systems until each is of a manageable complexity — the hierarchical design method [27].

1. A Hierarchical Design

Imagine that one wants to carry out the computation of a matrix multi-

plication; there are many possible ways to do it. One possible implementation of this function is by the algorithm shown in Figure 1–1 [22].

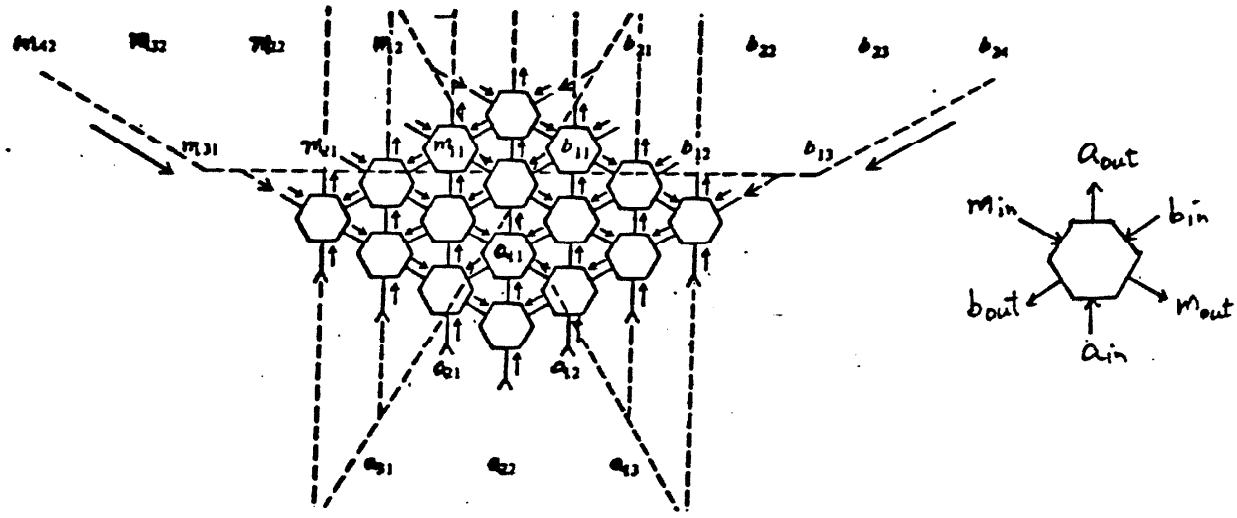


Figure 1–1 A Systolic Algorithm

The elementary building block has three inputs and three outputs, each of which is a bounded integer. Each element performs $a_{out} = m_{in} \times b_{in} + a_{in}$, $m_{out} = m_{in}$ and $b_{out} = b_{in}$. How each element is implemented is not of concern here, only the behavioral description of each of the elements used in designing and reasoning about the systolic algorithm. The complexity and performance of the algorithm are also discussed based on these measurements of each element. The area required by the algorithm is discussed using the area of each element as a unit, and the time required are measured using that required by each element as a unit. Once the design of this algorithm is completed, that is, once it is verified to be correct and to satisfy the requirements in performance, one can move on and focus on the design of each individual element.

Such an element can be implemented by serial operations on each bit of a binary number, or by concurrent operations on a word that stores the binary number. A possible bit-serial implementation [25],[44] is shown in Figure 1–2; three sequences of input bits are shifted into the pipeline and the result sequences come out the other end.

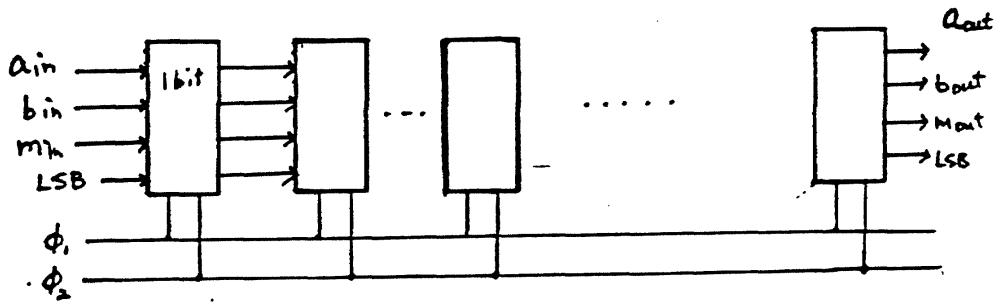


Figure 1–2 A Pipelined Algorithm

The elementary building block is now a half adder with some shift registers. Some effort may be spent on how to minimize the total delay so that the maximum throughput is possible. Boolean algebra is used in verifying the correctness of the algorithm. When the design at this level is completed, one moves on to a detailed implementation of each element.

Again a particular implementation is proposed, and the logic circuit shown in Figure 1–3 is an "algorithm" describing the design.

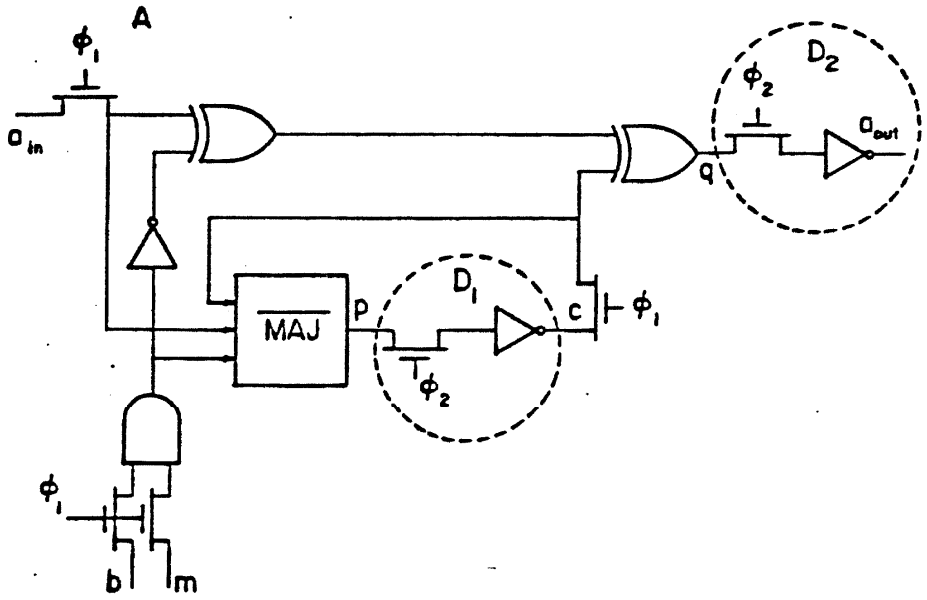


Figure 1–3 One bit inner product circuit — an algorithm

At this level, switching logic elements are used. In turn, these logic elements are implemented in a certain technology. Transistors, capacitors, etc., are used to implement these logic functions as shown in Figure 1–4. At this level, algebra of signals [3] will be used.

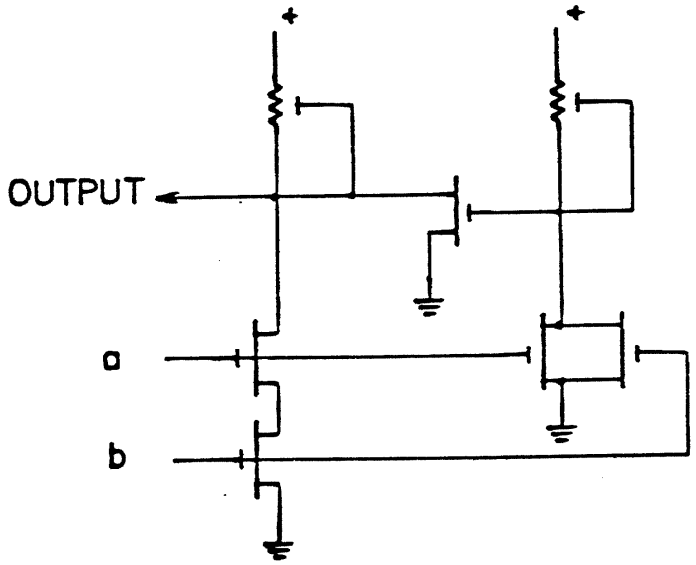


Figure 1–4 A transistor network — an algorithm

Eventually, when the design is completed, it will be realized in a physical domain. By partitioning the design into these levels, the designer needs only to concentrate on one level at a time. The details of lower levels are completely hidden.

2. Interplay of Space and Time

In each of the algorithms chosen above, the exact location and time step for the arrival of data at a processing element are important. The interplay of space and time does not occur in a program in a conventional "high-level" language since only one thing happens at a time, and where each item of data is physically stored is not important. This is not the case in the design of VLSI systems. Data that are far apart cost more energy and take more time to access, since a longer wire means larger capacitance and resistance. What is considered a good algorithm in a traditional language is not necessarily good when physical cost is taken into account. For example the algorithm Quicksort has performance advantage $n \log n$ over the n^2 required by the Interchange sort algorithm. Yet Quicksort involves swapping data that are arbitrarily far apart; each unit of cost is as expensive as accessing data that are farthest away. On the other hand, interchange sort only swaps data that are next to each other, thus each unit of cost is that involved in accessing only the neighboring elements [41]. This locality of communication is the key to an algorithm that is amenable to VLSI implementation. Hence VLSI is an environment in which both space and time need to be considered for design, verification and analysis of algorithms.

3. Notation for Describing VLSI Systems

So far the above examples of algorithms for VLSI are described by pictures. It is possible to reason about an algorithm informally using pictures

when it is relatively simple. A formal notation will not only be helpful in reasoning about the design, but can also be used to generate simulation and drive the compilation or assembly of physical layouts. Moreover, it is the first step towards an automatic verification tool (theorem prover) for VLSI systems. There have been many proposals for specification of concurrent systems. They differ in expressiveness, treatment of semantics, range of applicability, and programming styles — to name a few of them, data flow equations by Kahn[20], Actor Systems by Hewitt and Baker[15], its semantics of nondeterministic nature by Clinger [10], CSP by Hoare [16], CCS by Milner [30], Trace Theory by Rem, van de Snepscheut and Udding [34], and work focused on hardware description by Gordon [13], Cardelli [7], and Milne [28]. A special class of concurrent systems — linear systems — can be specified using z-transform notation. The work by Johnsson et. al.[19], illustrates not only verification but synthesis of these systems. However, none of the notations has the combination of all the following properties to allow specification of general concurrent VLSI systems.

Formal Semantics. With the advent of computer aided design tools for VLSI circuits, specifying VLSI designs in an *executable* form has become well-accepted. However, the specification must be *manipulable* as well for the following reason: the hierarchical design method demands proper interfaces among pieces of the design at each level as well as consistency between the abstract model at one level and the implementation at the level below. At a very low level, where the inputs and internal states of a sub-system are only a few bits, consistency checking can be done by exhaustively verifying all possible cases. For any larger sub-system, the consistency check amounts to verification of a program, where the program corresponds to the specification of the sub-system. Therefore, the specification language must have a formal semantics so that the specification of a given design can be formally verified.

Applicability. Another important property of a language for VLSI systems is its applicability to objects of all levels. It must be able to describe a network of processes as well as a transistor network. It must be capable of describing a realistic system in a simple way and at the same time must be amenable to a simple treatment of semantics. A language is of no use if either the description or the semantics is insurmountably complicated.

Referential Transparency. Functional (or applicative) languages, different from imperative (or assignment-based) languages, have the property that different occurrences of the same expression in a program always have the same value. A program written in a functional language is therefore free of side-effects and easier to reason about. Functional language also describes the following kind of concurrency automatically: functions g and h in $f(g(x,y),h(x,z))$ are independent due to the referential transparency of variable x and therefore can be evaluated concurrently.

History Sensitivity. Variables in an imperative language retain the state of a computation so that it can be used to affect the behavior later in the computation. This property is especially important in describing real-time systems where input to a program is a stream of data taken by the program as the computation proceeds rather than given initially as a set of data. It is very difficult to describe concurrent systems such as architectures for signal processing without explicit modification of state. A functional description for such applications becomes very cumbersome.

In this thesis, a methodology for specifying and verifying VLSI systems is presented. The focus is on deterministic concurrent systems rather than non-deterministic systems in general. The notation allows description of non-deterministic systems, but in common with other treatments lacks a

completely satisfactory treatment of their semantics. Following Carnap's *Axiom Systems for Physics*[6], explicit space and time parameters are used to describe a given concurrent system. With explicit time as a parameter, change of state can be expressed by state transition functions in the manner of [1]. Hence the language is history sensitive and at the same time allows the clarity of functional programming. The same notation applies to systems ranging from the level of transistors up to the level of communicating processes. Fixed-point semantics [37] is used for abstracting the behavior of system from its implementation. The consistency between an implementation and its description can therefore be verified. The framework can be viewed as a concurrent programming notation when describing communicating processes; a hardware description notation when specifying integrated circuits.

4. An Overview

In the next chapter, a model of computation for concurrent systems is first presented. It differs from "Communicating Sequential Processes" [16] in two important aspects.

(1) It separates the deterministic concurrent model from a more general nondeterministic one. The former is presented first and the extension to the latter in Chapter 6.

(2) An applicative state transition in the sense of [1] is introduced so that a functional style of programming is possible.

The syntax and semantics of the language CRYSTAL are introduced. The description of a system is called a space-time algorithm since space and time are explicit in the description. Some knowledge about Scott's theory of computable functions is assumed. In the third chapter, the systolic and

pipelined algorithms are described and proofs of their correctness are given. Since the class of asynchronous algorithms is quite different from systems with global clocks, two versions of the systolic algorithm are presented and compared.

In Chapter 4, formal semantics for an arbitrary transistor network is given. In Chapter 5, an "interpreter" for space-time algorithms — a hierarchical simulator — for VLSI systems is presented. Finally, some extensions of the work and future directions for the research are discussed.

Chapter 2

A Framework for Expressing Concurrency

1. Model of Computation

The model consists of a collection of *processes*[11], [17], [16]. Each process has

(i) a *control state* register for determining the communication of the process with other processes,

(ii) a data store,

(iii) the machinery for computing a *state transition function* [1].

(iv) input ports and output ports. A port is *filled* in the sense that a place in a Petri-net [33], [32] is filled by a token and *emptied* as a token is removed from a place when the corresponding transition is fired. Without loss of generality, we assume that each port can be filled only by one data item at any instant, or in Petri-net term, each place has at most one token at a time. This assumption is equivalent to that each port is a queue of finite length n (or at most n tokens in a any place of a given Petri net).

To describe the relationship among processes, coordinate systems are used.

(v) Each process is located in a space coordinate system where each coordinate is taken from a countable set.

(vi) It is often convenient to use a global time coordinate to index the operations occurring in the ensemble when there exists a total ordering of these operations. In general, operations occurring in the ensemble of processes are concurrent and cannot be totally ordered. Therefore, each process has a local time coordinate, taken from a countable well ordered set, that is used to index the operations each process has performed. Relationships among these local time coordinates of the processes must be derived for verifying the correctness of an algorithm.

(vii) The relationship among processes is established by identifying input port of process s_1 at its t_{s_1}'th operation with the output port of process s_2 at its t_{s_2}'th operation. The counting of the operations is defined below.

A process operates according to the following procedure.

(viii) The state of the control state register is used to select a set of input ports. The state of this register is used only in this way and is not used as an argument to a state transition function.

(ix) If all of the selected input ports are filled, we say that communication is established. The process now starts an operation, called t'th invocation, consisting of 1) emptying the selected input ports and 2) evaluating the state transition function (firing a transition as in Petri-net), called the t'th *invocation* of the process where t is the time coordinate of the process. If some input ports are not filled, the process waits until they are.

(x) Data in the selected input ports and the state in the data store are arguments to the state transition function. After the function is evaluated, the result is used to update the data store and control state register and to fill the designated output ports. If some of the designated output ports are not emptied, the process waits until they are emptied and then fills them. After all designated output ports are filled, the t'th invocation is completed and the process starts step (viii) again for the $t+1$'th invocation where $t+1$ denotes the next element in the well ordered set for the time coordinate.

2. Universality of the Model

Such a collection of processes is clearly as powerful as a Turing machine since the machinery in each process can be a Turing Machine if there is no bound on the size of the data store. The interesting case is that each of the processes is a finite automata. Such a system is shown to be universal by von Neumann [42]. To illustrate this model, we show how to simulate a Turing machine using an ensemble of an ***unbounded*** number of finite processes (each of the parts (i) — (iii) is finite). We construct an "object-oriented" Turing machine O using a collection of finite processes according to a given Turing machine T.

Let each process have the finite state machine of Turing machine T and a data store that corresponds to one tape square of T. The control state register has three possible states, "left", "right" and "self". The processes are arranged as a linear array. We can index the processes by using integers. The particular process corresponding to the square which is initially under the read head has space coordinate 0 and is called p_0. The processes to the left of p_0 have coordinate $-1, -2, \ldots$,etc, and those to the right have coordinate $1, 2, \ldots$,etc. A global time coordinate is used since the events occurring in this

machine are totally ordered. Each process has a pair of input and output ports to each of its two neighbors and a pair of "self-linked" input and output ports. Connections between neighboring processes are made by identifying the input port of one process with the corresponding output port of the neighbor for all t where t is the time coordinate. The "self-linked" input port is identified with the corresponding output port for all t. Each of these processes corresponds to a square of the tape in machine T and the finite number of symbols inscribed on the tape initially are put in the data store of each corresponding process.

Initially, the control state register of process p_0 is set to "self". The control state register of all the processes to the left of p_0 is set to be "right" and those to the right of p_0 be "left". Machine O is started from the environment by filling the "self-linked" input port of process p_0 with q_0, the initial state of T. After s_0 in the data store (symbol on the corresponding square in T) and input q_0 (current state in T) are applied to the state transition function (finite state machine in T), process p_0 updates the data store with the new symbol s_1 (as T would write on the tape), sends q_1 (the new state of T) to either the left, the right, or the "self-linked" output ports according to the direction of read head movement in T. The control state is updated as "right", "left" or "self" according to the head moving left, right, or not moving, respectively. An invocation is complete at this point and the process which receives the above new symbol in its input port is ready for action. Note that in machine O, only one process is active at any instant since there is only one tape square being read at any instant in T. This sequentiality allows us to order the invocations of processes in a global time frame. It is interesting to see that what is stored as state in T becomes input/output in O and vice versa.

The computation of O consists of a sequence of the above basic events which correspond to the basic moves that constitute the computation of T. Machine O either proceeds indefinitely as machine T would or terminates after some finite number of events and the result of the computation is the symbols in the data store in each process. It is clear from this construction that machine O simulates Turing Machine T. Hence we have shown that our model of computation is as powerful as a Turing machine. A simulator for any system modeled as an ensemble of processes will be described in Chapterb 5. It is implemented in a conventional programming environment and therefore completes the demonstration of the equivalence of the two models.

3. The Language CRYSTAL*

The language and its semantics are based on the typed λ-calculus version [29] of Scott's theory of computable functions [37]. Each of the parts or operations of the computation model above is described by either a constant value, a variable, a function or a function of functions.

Data Types. There are various data types for state and inputs/outputs for a wide range of systems of interests. Examples of data types are

(i) The set of analog voltage values $\mathcal{A} \equiv \{ a : 0 \leq a \leq V \}$: a subset of the set of real numbers.

(ii) The set of boolean values $\mathcal{B} \equiv \{ \hat{0}, \hat{1} \}$.

(iii) The set of n-bit words in 2's complement representation $\{ i : -2^{(n-1)} \leq i < 2^{(n-1)} \}$: a subset of the set of integers.

State transition functions. According to the model of computation, a state transition function is the basic unit used in constructing a system.

*CRYSTAL stands for "Concurrent Representation of Your Space-Time Algorithm

Depending on the systems of interest, different functions are used as a primitive state transition functions. In designing a VLSI system, a primitive can be an analog model of a transistor when detailed electrical characteristics of the system are desired. A switch level model of a transistor is used if the logic values computed by the circuit are desired. An adder is the primitive, for example, when a multiplier is built. A state transition function can also be implemented as a collection of existing systems rather than given as a primitive, in which case it is called a composite state transition function. Its meaning will be precisely defined below.

In describing a primitive state transition function, say, a model of a transistor, functions like addition, minimum, maximum on subsets of integers are used. These functions are primitives of the language and must not be confused with those of the system under construction. We may well use the plus function (which models parts of the behavior of a transistor) to construct a piece of machinery for computing the plus function (which models an n-bit adder, for example).

Let $\mathbf{x} \equiv (x_1, x_2, \ldots, x_m)$ denote the arguments of a state transition function f where $x_i \in D_i$ (the data type of x_i), $i = 1, 2, \ldots, m$.

$$\begin{aligned} f &: D_1 \times D_2 \times \cdots \times D_m \to D^n, \\ f(\mathbf{x}) &= (f_1, f_2, \ldots, f_n)(\mathbf{x}) \end{aligned} \tag{1}$$

Each component f_i of such a function is an element of $[D_1 \times D_2 \times \cdots \times D_m \to D]$; in the syntax below, we call each component a state transition function.

Coordinates. To express the relationship among invocations of processes in the space-time coordinate system, or to express different state transition functions (or different relationships of invocations) at different points in the coordinate system it is necessary to use functions on these coordinates. These functions are primitives of the language and like the plus

function, must not be confused with elements of the system being described. Although expressions of coordinates are not part of the computation of the system being designed, they are part of how the system is going to be constructed (space coordinates) and used (time coordinates), i.e., what timing discipline is imposed. Examples of such data types:

(i) The discrete time domain $\mathcal{T} \equiv \{0, 1, 2, \ldots\}$ (the set of non-negative integers).

(ii) The discrete space domain $\mathcal{S} \equiv \{(x, y) : 0 \leq x < n, 0 \leq y < n\}$, where x, y are integers.

(iii) The set of k-tuples $\mathcal{E} \equiv \{(e_1, e_2, \ldots, e_{k-1}, e_k)\}$ where $e_1, e_2, \ldots, e_k$ are space and time coordinates. This class of data types is called the space-time domain.

Data Streams. Each process is a point in a space coordinate system and each invocation of a process is a point in a space-time coordinate system where the space coordinates are the same as the corresponding space coordinates of the process and the time coordinate is local within the process. Control state, data, inputs and outputs of all processes as used in the invocation have the same coordinates as the invocation. The state and input/output values are defined in the space-time coordinates as the computation proceeds. They are expressed as unknown functions from the space-time domain to a certain value domain. In the beginning of the computation, only the initial state and initial inputs are defined. As the computation continues, more state and inputs/outputs become defined in the space-time domain; the computation ends when no more of them are defined. Each of these unknowns is called a data stream.

Space-time Algorithm. An algorithm describing a system consists of the description of the computational part (applying state and input/output

to state transition functions) and the communicative part (equating or identifying an input with an output at another point in the space-time domain). The result is a system of recursion equations in the space-time coordinates.

4. The Syntax of CRYSTAL

In the following, syntactic objects are capitalized and the semantic objects are in lower case. IN and OUT are names of data streams. Construct ⟨PTERM⟩ specifies the computation part of an algorithm and ⟨CTERM⟩ the relationships (connections) among invocations. ⟨ETERM⟩ specifies an expression in the space-time domain and ⟨DTERM⟩ that in the value domain. $A_1, A_2, \ldots, B_1, B_2, \ldots$ denote fixed constants in the space-time domain and the value domain, respectively. $F_1, \ldots, F_j$, $F'_1, \ldots, F'_j$, $G_1, G_2, \ldots, G_j$, and $H_1, \ldots, H_j$ denote fixed primitive functions over various domains. An example of them would be a "if-then-else" function.

⟨SPACE-TIME ALGORITHM (STA)⟩ ←

$$(OUT_1(S,T_s),\ldots,OUT_n(S,T_s),IN_1(S,T_s),\ldots,IN_m(S,T_s)) = (\langle \text{PTERM}\rangle_1,\ldots,\langle \text{PTERM}\rangle_n,\langle \text{CTERM}\rangle_1,\ldots,\langle \text{CTERM}\rangle_m)$$

⟨PTERM⟩ ←⟨ETERM⟩

$$\begin{aligned}
&\mid \langle \text{STATE TRANSITION FUNCTION}\rangle(IN_1(S,T_s),\ldots,IN_m(S,T_s))^{\dagger}\\
&\mid F_1(\langle \text{PTERM}\rangle_1,\langle \text{PTERM}\rangle_2,\ldots,\langle \text{PTERM}\rangle_{l_1})\\
&\quad\vdots\\
&\mid F_j(\langle \text{PTERM}\rangle_1,\langle \text{PTERM}\rangle_2,\ldots,\langle \text{PTERM}\rangle_{l_j})
\end{aligned}$$

⟨ETERM⟩ ←

$$\begin{aligned}
&\mid A_1 \mid A_2 \mid \cdots\\
&\mid S \mid T_s\\
&\mid G_1(\langle \text{ETERM}\rangle_1,\langle \text{ETERM}\rangle_2,\ldots,\langle \text{ETERM}\rangle_{l_1})\\
&\quad\vdots\\
&\mid G_j(\langle \text{ETERM}\rangle_1,\langle \text{ETERM}\rangle_2,\ldots,\langle \text{ETERM}\rangle_{l_j})
\end{aligned}$$

†State transition functions can have various numbers of bound variables, for simplicity, we always use m of them

$$\langle \text{STATE TRANSITION FUNCTION (STF)} \rangle \leftarrow \langle \text{PRIMITIVE STF} \rangle \mid \langle \text{COMPOSITE STF} \rangle^{\dagger}$$

$$\langle \text{PRIMITIVE STF} \rangle \leftarrow \lambda(X_1, X_2, \ldots, X_m).\langle \text{DTERM} \rangle$$

$$\begin{aligned}\langle \text{DTERM} \rangle \leftarrow & B_1 \mid B_2 \mid \cdots \\ & \mid X_1 \mid X_2 \mid \cdots X_m \\ & \mid H_1(\langle \text{DTERM} \rangle_1, \langle \text{DTERM} \rangle_2, \ldots, \langle \text{DTERM} \rangle_{l_1}) \\ & \vdots \\ & \mid H_j(\langle \text{DTERM} \rangle_1, \langle \text{DTERM} \rangle_2, \ldots, \langle \text{DTERM} \rangle_{l_j})\end{aligned}$$

$$\begin{aligned}\langle \text{CTERM} \rangle \leftarrow & \langle \text{ETERM} \rangle \\ & \mid OUT_1(\langle \text{ETERM} \rangle_s, \langle \text{ETERM} \rangle_t) \\ & \vdots \\ & \mid OUT_n(\langle \text{ETERM} \rangle_s, \langle \text{ETERM} \rangle_t) \\ & \mid X'_1 \mid X'_2 \mid \cdots \mid X'_{m'} \\ & \mid F'_1(\langle \text{CTERM} \rangle_1, \langle \text{CTERM} \rangle_2, \ldots, \langle \text{CTERM} \rangle_{l_1}) \\ & \vdots \\ & \mid F'_j(\langle \text{CTERM} \rangle_1, \langle \text{CTERM} \rangle_2, \ldots, \langle \text{CTERM} \rangle_{l_j})\end{aligned}$$

5. The Semantics of CRYSTAL

The meaning of ⟨SPACE-TIME ALGORITHM (STA)⟩ is a continuous functional

$$\Psi(OUT_1, \ldots, OUT_n, IN_1, \ldots, IN_m) \equiv$$
$$(\Sigma[\![\langle \text{PTERM} \rangle_1,]\!] \ldots, \Sigma[\![\langle \text{PTERM} \rangle_n]\!], \Sigma[\![\langle \text{CTERM} \rangle_1]\!], \ldots, \Sigma[\![\langle \text{CTERM} \rangle_m]\!])$$

where the semantic function Σ maps from the syntactic catagory of terms

†An algorithm can be built from primitive state transition functions or another algorithm which implements a state transition function. This construct will be given in the last section when the behavior of an algorithm is defined.

to $[Env \rightarrow S + \mathcal{T} + \mathcal{D}]$, where Env is the environment and $\rho \in Env$. This semantic function is defined inductively as follows.

(i) Constants.

$$\Sigma[\![A_i]\!]\rho \equiv a_i, \quad a_i \in S + \mathcal{T}$$
$$\Sigma[\![B_i]\!]\rho \equiv b_i, \quad b_i \in \mathcal{D}.$$

(ii) Variables.

$$\Sigma[\![S]\!]\rho \in S,$$
$$\Sigma[\![T_s]\!]\rho \in \mathcal{T},$$
$$\Sigma[\![X_i]\!]\rho \in \mathcal{D}.$$
$$\Sigma[\![X_i']\!]\rho \in \mathcal{D}.$$

(iii) Functions.

$\Sigma[\![G(\langle \text{ETERM}\rangle_1, \ldots, \langle \text{ETERM}\rangle_l)]\!] \equiv g(\Sigma[\![\langle \text{ETERM}\rangle_1]\!], \ldots, \Sigma[\![\langle \text{ETERM}\rangle_l]\!])$
where g is some fixed continuous function in $[(S + \mathcal{T})^l \rightarrow S + \mathcal{T}]$
$\Sigma[\![H(\langle \text{DTERM}\rangle_1, \ldots, \langle \text{DTERM}\rangle_l)]\!] \equiv h(\Sigma[\![\langle \text{DTERM}\rangle_1]\!], \ldots, \Sigma[\![\langle \text{DTERM}\rangle_l]\!])$
where h is some fixed continuous function in $[\mathcal{D}^l \rightarrow \mathcal{D}]$
$\Sigma[\![F(\langle \text{PTERM}\rangle_1, \ldots, \langle \text{PTERM}\rangle_l)]\!] \equiv f(\Sigma[\![\langle \text{PTERM}\rangle_1]\!], \ldots, \Sigma[\![\langle \text{PTERM}\rangle_l]\!])$
where f is some fixed continuous function in $[(S + \mathcal{T} + \mathcal{D})^l \rightarrow S + \mathcal{T} + \mathcal{D}]$
$\Sigma[\![F'(\langle \text{CTERM}\rangle_1, \ldots, \langle \text{CTERM}\rangle_l)]\!] \equiv f'(\Sigma[\![\langle \text{CTERM}\rangle_1]\!], \ldots, \Sigma[\![\langle \text{CTERM}\rangle_l]\!])$
where f' is some fixed continuous function in $[(S + \mathcal{T} + \mathcal{D})^l \rightarrow S + \mathcal{T} + \mathcal{D}]$

The least fixed-point of the functional Ψ is defined in terms of the least fixed-point of each component of Ψ, denoted by $(OUT_1^\infty, \ldots, OUT_n^\infty, IN_1^\infty, \ldots, IN_m^\infty)$. All constructs above are continuous, which is proved by induction on the structure of terms by using the closure property of continuous functions under composition, λ-abstraction and the fixed-point operation. The proofs can be found in [29].

6. The Behavior of an Algorithm

The function an algorithm computes, the least fixed-point of the functional, is a function from the space and time domain to the value domain. If

an algorithm is to be treated as a black box when used to construct some other algorithms, we must describe outputs at the end of a computation in terms of inputs at the beginning of the computation. This behavioral description is obtained from the least fixed-point by the following:

⟨RESULT OF STA⟩ ←

$$(Y'_1,\ldots,Y'_{n'}) \equiv \langle\text{BEHAVIOR OF STA}\rangle(IN^{\infty}_{j_1}(\langle\text{IPORT}\rangle_1),\ldots, IN^{\infty}_{j_{m'}}(\langle\text{IPORT}\rangle_{m'}))$$
$$\text{where } j_1,\ldots,j_{m'} \in \{1,2,\ldots,m\}$$

⟨BEHAVIOR OF STA⟩ ← (⟨COMPOSITE STF⟩$_1$, ..., ⟨COMPOSITE STF⟩$_{n'}$)

⟨COMPOSITE STF⟩ ← $\lambda(X'_1,\ldots,X'_{m'}).OUT^{\infty}_j(\langle\text{OPORT}\rangle)$
where $j \in \{1,2,\ldots,n\}$

⟨OPORT⟩ ← (A_s, A_t)
where $\Sigma[\![A_s]\!] = a_s \in S$ and $\Sigma[\![A_t]\!] = a_t \in T$.

⟨IPORT⟩ ← (A_s, A_t)
where $\Sigma[\![A_s]\!] = a_s \in S$ and $\Sigma[\![A_t]\!] = a_t \in T$.

The semantics of ⟨BEHAVIOR OF STA⟩ is an n'-tuple function from $D^{m'}$ to $D^{n'}$. Thus the system described by the algorithm can be abstracted as a system of state transition functions that maps inputs and current states $(X'_1,\ldots,X'_{m'})$ to outputs and next states $(Y'_1,\ldots,Y'_{n'})$. It therefore can serve as a primitive building block (⟨COMPOSITE STF⟩) for constructing a more complex system.

7. A Simple Example of a Space-Time Algorithm

The following is a very simple space-time algorithm which corresponds to the program that computes the factorial function in an assignment based language. Given an input a, this program computes fac as the result.

$count \leftarrow a, fac \leftarrow 1$
while $count > 0$ **do**
begin $fac \leftarrow fac \times count; count := count - 1$; **end**;

For any $t \geq a$, in the following corresponding space-time algorithm, $\mathbf{fac}(t) = a!$. Since this algorithm is sequential, only one process is needed and the space domain degenerates to one point. Let $\mathcal{T} = 0, 1, 2, \ldots$ be the time domain.

$$\mathbf{count}(t) = \begin{cases} t = 0 \rightarrow a \\ t > 0 \rightarrow \begin{cases} \mathbf{count}(t-1) > 0 \rightarrow \mathbf{count}(t-1) - 1 \\ \mathbf{count}(t-1) \leq 0 \rightarrow \mathbf{count}(t-1) \end{cases} \end{cases}$$

$$\mathbf{fac}(t) = \begin{cases} t = 0 \rightarrow 1 \\ t > 0 \rightarrow \begin{cases} \mathbf{count}(t-1) > 0 \rightarrow \mathbf{fac}(t-1) \times \mathbf{count}(t-1) \\ \mathbf{count}(t-1) \leq 0 \rightarrow \mathbf{fac}(t-1) \end{cases} \end{cases} \tag{2.1}$$

Data streams in this example are **count** and **fac**. Both $\mathbf{count}(t)$ and $\mathbf{fac}(t)$ are of type $\mathcal{N} \equiv \{0, 1, 2, \ldots\}$. Three state transition functions in the algorithm are $f(x_1, x_2) = (a, 1)$ where each component is a constant function, $g(x_1, x_2) = (x_1 - 1, x_2 \times x_1)$, and $h(x_1, x_2) = (x_1, x_2)$. Notice that in the above algorithm, **count** is used as a variable for keeping track of the number to be multiplied to the partial result. Why do we not write the algorithm as the following equation where the time coordinate t is used in the computation of factorial a?

$$\mathbf{fac}(t) = \begin{cases} t = 0 \rightarrow 1 \\ t > 0 \rightarrow \begin{cases} t \leq a \rightarrow \mathbf{fac}(t-1) \times t \\ t > a \rightarrow \mathbf{fac}(t-1) \end{cases} \end{cases}$$

The reason is that t is not part of the implementation, it is only a reference frame for us to envision and reason about the computation. To see that the algorithm computes the factorial function, we claim that the following is the least fixed-point of (2.1).

$$\mathbf{count}^{\infty}(t) = \begin{cases} t > a \rightarrow 0 \\ t \leq a \rightarrow a - t \end{cases}$$

$$\mathbf{fac}^{\infty}(t) = \begin{cases} t > a \rightarrow a! \\ t \leq a \rightarrow \frac{a!}{(a-t)!} \end{cases} \tag{2.2}$$

By induction on t, (2.2) can be shown to be the least solution of (2.1). Notice that for all $t > a$, both $\mathbf{count}(t) = \mathbf{count}(t-1)$ and $\mathbf{fac}(t) = \mathbf{fac}(t-1)$. Thus the time domain can be restricted such that $t \leq a$. In general, a system reaches its *steady state* at t_{steady} and

$$t_{steady} = \max_{t_i}\{\mu t_i[\lambda s.(Stream_i(s,t) - Stream_i(s,t-1)) = \lambda s.0] : i \in \{1,2,\ldots,n\}\}$$

where μ is the minimalization operator and i is the number of data streams.

The behavior of the algorithm is $\lambda a.\mathbf{fac}(t_{steady}) = \lambda a.a!$, and thus the algorithm implements the factorial function correctly.

Chapter 3

Semantics of Systolic Arrays

Systolic structure is a term used to describe a class of computational arrays in which locality of communication is employed to achieve a high system throughput [21]. It has been applied to the areas of signal processing, pattern matching for data bases, parsing of formal languages, etc., (for references, see [21]). In the following, a formal description of the matrix multiplication algorithm by [22] is given with a proof of correctness. Since different timing schemes result in different descriptions and proofs, both the synchronous version and self-timed version are given. Kung's original algorithm assumes a global clock, i.e., every process performs an operation synchronously. The same algorithm with a different timing scheme, e.g., a self-timed [39] scheme can conceptually simplify the interaction of processes, the flow of data, and the initiation of the system. This simplicity results from the fact that the self-timed scheme assures that each process does not perform any operation until all the meaningful data items have reached the process. On the other hand, the self-timed scheme does not have any global control, the ordering of the system events is an emergent property of the local synchronization. Thus the specification of the relationships among invocations of processes has to be verified.

1. Matrix Multiplication on a Systolic Array

As shown in Figure 1–1, identical elements are interconnected into a

hexagonal array. Each element has three inputs and three outputs as indicated by the incoming and outgoing arrows, respectively. Such an element performs an inner product operation, i.e., $c_{out} = c_{in} + a_{in} \times b_{in}$, and transmits the other two inputs, i.e., $a_{out} = a_{in}$, $b_{out} = b_{in}$. Two matrices to be multiplied, A and B, and a matrix C for accumulating the partial results are fed into the array as shown in the figure. The final results will come out from the top of the array as shown.

Each element is represented by a process in our model. It has only one control state, namely, always choosing all input ports. It does not have any internal state but only takes inputs from input ports and writes to output ports, and therefore the data store is empty. The three components of the state transition function are described above. As shown in Figure 3–1, the 3-dimensional Cartesian coordinate system is chosen as the space coordinates because the symmetry of the data flow can be described by the dihedral group of order 3 [23]. The center of this hexagon (coordinates (0,0,0)) can be viewed as a corner of a cube; the hexagon covers the three faces that contain this corner. The relationship among the time coordinates of processes depends on the timing scheme.

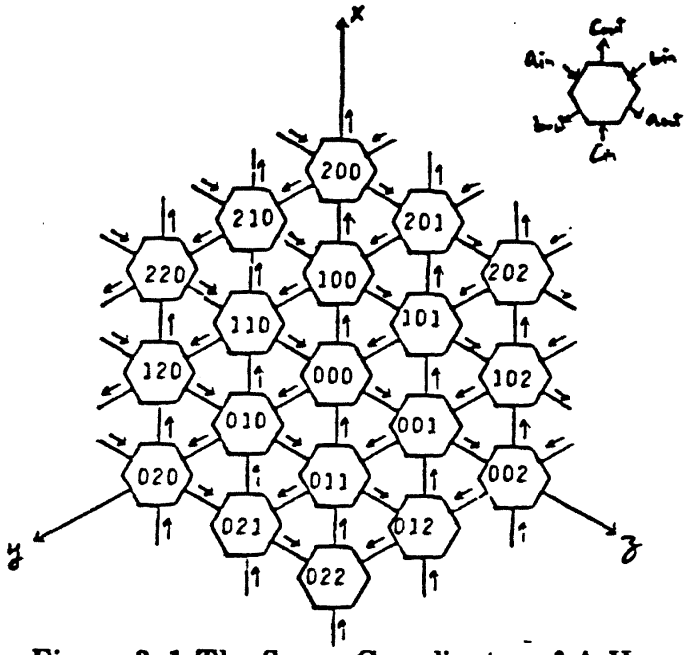


Figure 3–1 The Space Coordinates of A Hexagonal Array

2. A Synchronous System

In a synchronous system, a global clock triggers the operations of all elements at once. The clock period (cycle time) is the maximum of the amount of time necessary for each invocation to be completed. In this case, the time coordinates of all processes have a simple relationship: they are identical. Hence a global time coordinate is used for invocations of all processes.

The Algorithm. Let x, y, z be the space coordinates and t the time coordinate; they are non-negative integers. The space-time domain is defined using the the following expressions which indicate parts of the domain. Some of these expressions are shown in Figure 3–2. For example, φ_{xy} restricts the xy plane to an area within the specified bounds in the first quadrant.

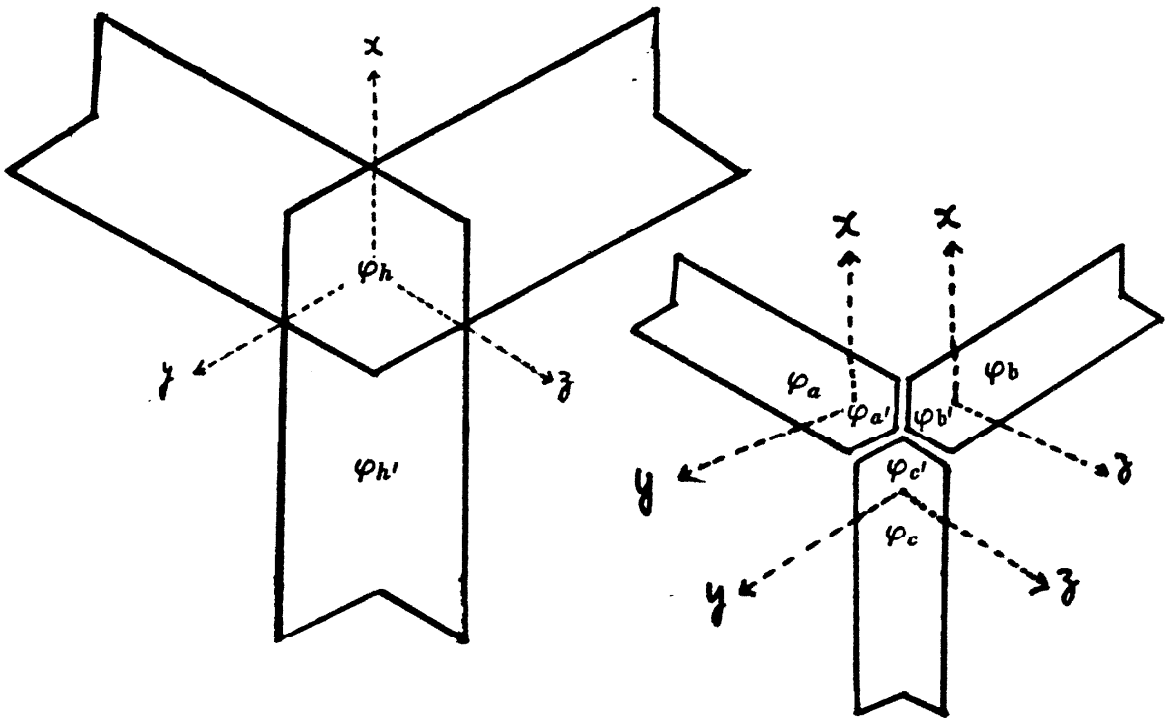


Figure 3–2 The initialization of a synchronous systolic array

Expressions correspond to quadrants, axis, and the origin:

$$\varphi_{xy} \equiv (n > x > 0) \wedge (n > y > 0) \wedge (z = 0)$$
$$\varphi_{yz} \equiv (n > y > 0) \wedge (n > z > 0) \wedge (x = 0)$$
$$\varphi_{zx} \equiv (n > z > 0) \wedge (n > x > 0) \wedge (y = 0)$$
$$\varphi_{x} \equiv (n > x > 0) \wedge (y = 0) \wedge (z = 0)$$
$$\varphi_{y} \equiv (n > y > 0) \wedge (z = 0) \wedge (x = 0)$$
$$\varphi_{z} \equiv (n > z > 0) \wedge (x = 0) \wedge (y = 0)$$
$$\varphi_{o} \equiv (x = 0) \wedge (y = 0) \wedge (z = 0)$$

The areas where elements of matrices are placed initially:

$$\varphi_a \equiv (x \geq 0) \wedge (y \geq 0) \wedge (|x - y| < n) \wedge ((x < 3n) \vee (y < 3n)) \wedge (z = 0)$$
$$\varphi_b \equiv (z \geq 0) \wedge (x \geq 0) \wedge (|z - x| < n) \wedge ((z < 3n) \vee (x < 3n)) \wedge (y = 0)$$
$$\varphi_c \equiv (y \geq 0) \wedge (z \geq 0) \wedge (|y - z| < n) \wedge ((y < 3n) \vee (z < 3n)) \wedge (x = 0)$$
$$\varphi_{h'} \equiv ((y \geq n) \vee (z \geq n)) \wedge (|y - z| < n) \wedge (y < 3n) \wedge (z < 3n) \wedge (x = 0)$$

Expressions which are unions of other expressions:

$\varphi_h \equiv \varphi_{xy} \vee \varphi_{yz} \vee \varphi_{zx} \vee \varphi_x \vee \varphi_y \vee \varphi_z \vee \varphi_o$ (the area within the hexagon)
$\varphi_{a'} \equiv \varphi_{yz} \vee \varphi_{zx} \vee \varphi_z$
$\varphi_{b'} \equiv \varphi_{yz} \vee \varphi_{xy} \vee \varphi_y$
$\varphi_{c'} \equiv \varphi_{xy} \vee \varphi_{zx} \vee \varphi_x$

The space-time domain:

$$\varphi_s \equiv \varphi_a \vee \varphi_b \vee \varphi_c$$
$$\varphi_t \equiv 0 < t \leq 4(n-1)+1$$

Let $\mathbf{a}_{in}$, $\mathbf{b}_{in}$, and $\mathbf{c}_{in}$ be the input data streams, and $\mathbf{a}_{out}$, $\mathbf{b}_{out}$ and $\mathbf{c}_{out}$ be the output data streams. The following are ⟨PTERM⟩'s which specify the computation (functional) part of the algorithm. The symbol $\perp$ means undefined.

$$\mathbf{a}_{out}(x,y,z,t) = \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow \mathbf{a}_{in}(x,y,z,t) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{b}_{out}(x,y,z,t) = \begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow \mathbf{b}_{in}(x,y,z,t) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{c}_{out}(x,y,z,t) = \begin{cases} \varphi_{h'} \rightarrow \mathbf{c}_{in}(x,y,z,t) \\ \varphi_h \rightarrow \mathbf{c}_{in}(x,y,z,t) + \mathbf{a}_{in}(x,y,z,t) \times \mathbf{b}_{in}(x,y,z,t) \\ else \rightarrow \perp \end{cases} \quad (2)$$

⟨CTERM⟩'s specify the connections.

$$\mathbf{a}_{in}(x,y,z,t) = \begin{cases} t = 1 \rightarrow \mathbf{a}_0(x,y,z) \\ t > 1 \rightarrow \begin{cases} \varphi_a \rightarrow \mathbf{a}_{out}(x+1,y+1,z,t-1) \\ \varphi_{a'} \rightarrow \mathbf{a}_{out}(x,y,z-1,t-1) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$\mathbf{b}_{in}(x,y,z,t) = \begin{cases} t = 1 \rightarrow \mathbf{b}_0(x,y,z) \\ t > 1 \rightarrow \begin{cases} \varphi_b \rightarrow \mathbf{b}_{out}(x+1,y,z+1,t-1) \\ \varphi_{b'} \rightarrow \mathbf{b}_{out}(x,y-1,z,t-1) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$\mathbf{c}_{in}(x,y,z,t) = \begin{cases} t = 1 \rightarrow \mathbf{c}_0(x,y,z) \\ t > 1 \rightarrow \begin{cases} \varphi_c \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1) \\ \varphi_{c'} \rightarrow \mathbf{c}_{out}(x-1,y,z,t-1) \\ else \rightarrow \perp. \end{cases} \end{cases} \tag{3}$$

By substituting (3) into (2), $\mathbf{a}_{in}, \mathbf{b}_{in}$, and $\mathbf{c}_{in}$ are eliminated. The space-time algorithm obtained is a system of recursion equations in $\mathbf{a}_{out}$, $\mathbf{b}_{out}$ and $\mathbf{c}_{out}$ and the initial inputs $\mathbf{a}_0(x,y,z)$, $\mathbf{b}_0(x,y,z)$ and $\mathbf{c}_0(x,y,z)$.

$$\mathbf{a}_{out}(x,y,z,t) = \begin{cases} t = 1 \rightarrow \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow \mathbf{a}_0(x,y,z) \\ else \rightarrow \perp \end{cases} \\ t > 1 \rightarrow \begin{cases} \varphi_a \rightarrow \mathbf{a}_{out}(x+1,y+1,z,t-1) \\ \varphi_{a'} \rightarrow \mathbf{a}_{out}(x,y,z-1,t-1) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$\mathbf{b}_{out}(x,y,z,t) = \begin{cases} t = 1 \rightarrow \begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow \mathbf{b}_0(x,y,z) \\ else \rightarrow \perp \end{cases} \\ t > 1 \rightarrow \begin{cases} \varphi_b \rightarrow \mathbf{b}_{out}(x+1,y,z+1,t-1) \\ \varphi_{b'} \rightarrow \mathbf{b}_{out}(x,y-1,z,t-1) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$
\mathbf{c}_{out}(x,y,z,t) = \begin{cases}
t = 1 \rightarrow \begin{cases} \varphi_{h'} \rightarrow \mathbf{c}_0(x,y,z) \\ \varphi_h \rightarrow \mathbf{c}_0(x,y,z) + \mathbf{a}_0(x,y,z) \times \mathbf{b}_0(x,y,z) \\ else \rightarrow \perp \end{cases} \\
t > 1 \rightarrow \\
\quad \begin{cases}
\varphi_{h'} \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1) \\
\varphi_h \wedge \varphi_c \rightarrow \\
\quad \begin{cases}
\varphi_a \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1) \\
\qquad + \mathbf{a}_{out}(x+1,y+1,z,t-1) \times \mathbf{b}_{out}(x,y-1,z,t-1) \\
\varphi_b \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1) \\
\qquad + \mathbf{a}_{out}(x,y,z-1,t-1) \times \mathbf{b}_{out}(x+1,y,z+1,t-1) \\
else \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1) \\
\qquad + \mathbf{a}_{out}(x,y,z-1,t-1) \times \mathbf{b}_{out}(x,y-1,z,t-1)
\end{cases} \\
\varphi_h \wedge \varphi_{c'} \rightarrow \\
\quad \begin{cases}
\varphi_a \rightarrow \mathbf{c}_{out}(x-1,y,z,t-1) \\
\qquad + \mathbf{a}_{out}(x+1,y+1,z,t-1) \times \mathbf{b}_{out}(x,y-1,z,t-1) \\
\varphi_b \rightarrow \mathbf{c}_{out}(x-1,y,z,t-1) \\
\qquad + \mathbf{a}_{out}(x,y,z-1,t-1) \times \mathbf{b}_{out}(x+1,y,z+1,t-1)
\end{cases} \\
else \rightarrow \perp
\end{cases}
\end{cases}
\tag{4}
$$

The Input and Output Mapping Functions. The above algorithm is defined in terms of the initial inputs $\mathbf{a}_0$, $\mathbf{b}_0$ and $\mathbf{c}_0$. To see that the above algorithm performs matrix multiplication, it is necessary to map the elements of matrices from the indices to the space-time coordinates (an input mapping function) and map the final output from the space-time coordinates to the structure of matrices (an output mapping function). A Matrix with elements from some value domain D is a function from the domain N^2 of index pairs to D where $N \equiv \{0, 1, \ldots, n-1\}$.

Given space-time coordinates, the following pairs of input mapping functions give the index pair (i, j) for a matrix.

$$(I_a, J_a), (I_b, J_b), (I_c, J_c) \in [\mathcal{E} \to \mathcal{N}^2]$$

$$I_a(x,y,z) \equiv \begin{cases} 2y - x \equiv 0 \pmod 3 \to \frac{2y-x}{3} \\ else \to \perp \end{cases} \qquad (6a)$$

$$J_a(x,y,z) \equiv \begin{cases} 2x - y \equiv 0 \pmod 3 \to \frac{2x-y}{3} \\ else \to \perp \end{cases}$$

$$I_b(x,y,z) \equiv \begin{cases} 2x - z \equiv 0 \pmod 3 \to \frac{2x-z}{3} \\ else \to \perp \end{cases} \qquad (6b)$$

$$J_b(x,y,z) \equiv \begin{cases} 2z - x \equiv 0 \pmod 3 \to \frac{2z-x}{3} \\ else \to \perp \end{cases}$$

$$I_c(x,y,z) \equiv \begin{cases} 2y - z \equiv 0 \pmod 3 \to \frac{2y-z}{3} \\ else \to \perp \end{cases} \qquad (6c)$$

$$J_c(x,y,z) \equiv \begin{cases} 2z - y \equiv 0 \pmod 3 \to \frac{2z-y}{3} \\ else \to \perp \end{cases}$$

The initial inputs are defined in terms of matrices ($A, B, C \in [\mathcal{N}^2 \to D]$) and the input mapping functions.

$$\mathbf{a}_0(x,y,z) \equiv \begin{cases} \varphi_a \to \begin{cases} (2y - x \equiv 0 \pmod 3) \wedge (2x - y \equiv 0 \pmod 3) \\ \qquad \to A(I_a(x,y,z), J_a(x,y,z)) \\ else \to 0 \end{cases} \\ \varphi_{a'} \to 0 \\ else \to \perp \end{cases}$$

$$\mathbf{b}_0(x,y,z) \equiv \begin{cases} \varphi_b \to \begin{cases} (2x - z \equiv 0 \pmod 3) \wedge (2z - x \equiv 0 \pmod 3) \\ \qquad \to B(I_b(x,y,z), J_b(x,y,z)) \\ else \to 0 \end{cases} \\ \varphi_{b'} \to 0 \\ else \to \perp \end{cases}$$

$$\mathbf{c}_0(x,y,z) \equiv \begin{cases} \varphi_c \to \begin{cases} (2y - z \equiv 0 \pmod 3) \wedge (2z - y \equiv 0 \pmod 3) \\ \qquad \to C(I_c(x,y,z), J_c(x,y,z)) \\ else \to 0 \end{cases} \\ \varphi_{c'} \to 0 \\ else \to \perp \end{cases} \qquad (7)$$

Output mapping functions (X_a, Y_a, Z_a, T_a), (X_b, Y_b, Z_b, T_b) and (X_c, Y_c, Z_c, T_c) in $[\mathcal{N}^2 \rightarrow \mathcal{E}]$ where

$$\begin{aligned} &X_a(i,j) \equiv \max(j-i,0), Y_a(i,j) \equiv \max(i-j,0), Z_a(i,j) \equiv n-1 \\ &X_b(i,j) \equiv \max(i-j,0), Y_b(i,j) \equiv n-1, Z_b(i,j) \equiv \max(j-i,0) \\ &X_c(i,j) \equiv n-1, Y_c(i,j) \equiv \max(i-j,0), Z_c(i,j) \equiv \max(j-i,0) \\ &T_a(i,j) \equiv T_b(i,j) \equiv T_c(i,j) \equiv n + min(i,j) + i + j \end{aligned} \tag{8}$$

The resulting matrices A', B' and C' are defined as

$$\begin{aligned} A'(i,j) &\equiv \mathbf{a}^{\infty}_{out}(X_a(i,j), Y_a(i,j), Z_a(i,j), T_a(i,j)) \\ B'(i,j) &\equiv \mathbf{b}^{\infty}_{out}(X_b(i,j), Y_b(i,j), Z_b(i,j), T_b(i,j)) \\ C'(i,j) &\equiv \mathbf{c}^{\infty}_{out}(X_c(i,j), Y_c(i,j), Z_c(i,j), T_c(i,j)) \end{aligned} \tag{9}$$

The Proof of Correctness. The above system of recursion equations and the input and output mapping functions must be shown to correctly implement the familiar identity and matrix functions, i.e.

Proposition 1:

$$\begin{aligned} A'(i,j) &= A(i,j) \\ B'(i,j) &= B(i,j) \\ C'(i,j) &= \sum_{k=0}^{n-1} A(i,k) \times B(k,j) + C(i,j) \\ &\quad \text{where } 0 \leq i < n, \quad 0 \leq j < n \end{aligned}$$

The first step towards the proof is to verify the following function $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ to be the the least fixed point $(\mathbf{a}^{\infty}_{out}, \mathbf{b}^{\infty}_{out}, \mathbf{c}^{\infty}_{out})$ of the space-time algorithm (4). Given any point in the space-time domain, this function gives the values computed by the algorithm in terms of the initial inputs $\mathbf{a}_0$, $\mathbf{b}_0$ and $\mathbf{c}_0$.

Proposition 2:

$\mathbf{a}^{\infty}_{out} = \mathbf{a}$, $\mathbf{b}^{\infty}_{out} = \mathbf{b}$, and $\mathbf{c}^{\infty}_{out} = \mathbf{c}$ where

$$\mathbf{a}(x,y,z,t) \equiv \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow \\ \qquad \mathbf{a}_0(x + \max(t-1-z,0), y + \max(t-1-z,0), \\ \qquad\qquad \max(z-(t-1),0)) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{b}(x,y,z,t) \equiv \begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow \\ \quad \mathbf{b}_0(x+\max(t-1-y,0), \max(y-(t-1)), \\ \quad\quad z+\max(t-1-y),0) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{c}(x,y,z,t) \equiv \begin{cases} \varphi_c \vee \varphi_{c'} \rightarrow \\ \quad \mathbf{c}_0(\max(x-(t-1),0), y+\max(t-1-x,0), \\ \quad\quad z+\max(t-1-x,0)) \\ \quad\quad +S_1+S_2 \\ else \rightarrow \perp \end{cases} \tag{10}$$

where

$$S_1 \equiv \sum_{k=K_1}^{0} \mathbf{a}_0(x+k+\max(U_2,0), y+\max(U_2,0), \max(-U_2,0)) \\ \times \mathbf{b}_0(x+k+\max(U_1,0), \max(-U_1,0), z+\max(U_1,0))$$

and

$$S_2 \equiv \sum_{k=0}^{K_2} \mathbf{a}_0(\max(V_2,0), y+k+\max(V_2,0), \max(-V_2,0)) \\ \times \mathbf{b}_0(\max(V_1,0), \max(-V_1,0), z+k+\max(V_1,0)),$$

and

$$U_1 \equiv t-1+k-y, \quad U_2 \equiv t-1+k-z,$$
$$V_1 \equiv (t-1-x-k)-(y+k), \quad V_2 \equiv (t-1-x-k)-(z+k),$$
$$K_1 \equiv 1-\min(x,t-1), \quad K_2 \equiv \min(n-1-y, n-1-z, t-1-x)$$

Proof :

The proof is by induction on t for each phase of a "computational wave". The significance of defining phase is that the partial result of a given output is carried by a wavefront of a fixed phase. Let $\phi_i(x,y,z,t) = x+y+z+2t$ be the phase for the incident wave and $\phi_r(x,y,z,t) = x+y+z-t$ be that of the reflected wave.

First we show that $\mathbf{a}_{out}^{\infty} = \mathbf{a}$ and similarly, $\mathbf{b}_{out}^{\infty} = \mathbf{b}$. If not $\varphi_a \vee \varphi_{a'}$ then from (4) and (10) $\mathbf{a}_{out}^{\infty} = \perp = \mathbf{a}$. If $\varphi_a \vee \varphi_{a'}$ then

(i) If $t = 1$, since $z \geq 0$, $\max(z-(t-1),0) = z$ and $\max(t-1-z,0) = 0$. Therefore

$$\begin{aligned}\mathbf{a}^{\infty}_{out}(x,y,z,1) & \\ &= \mathbf{a}_0(x,y,z) \qquad \text{by (4)} \\ &= \mathbf{a}(x,y,z,1) \qquad \text{by (10)}\end{aligned}$$

(ii) $(t > 1) \wedge \varphi_a$.

$$\begin{aligned}\mathbf{a}^{\infty}_{out}(x,y,z,t) &= \mathbf{a}^{\infty}_{out}(x+1,y+1,z,t-1) \qquad \text{by (4)} \\ &= \mathbf{a}(x+1,y+1,z,t-1) \\ &\quad \text{by induction hypothesis since} \\ &\qquad \phi_i(x+1,y+1,z,t-1) = \phi_i(x,y,z,t) \\ &= \mathbf{a}_0((x+1)+(t-1)-1,(y+1)+(t-1)-1,0,0) \qquad \text{by (10)} \\ &= \mathbf{a}_0(x+t-1,y+t-1,0,0) \\ &= \mathbf{a}(x,y,z,t)\end{aligned}$$

(iii) $(t > 1) \wedge \varphi_{a'}$.

$$\begin{aligned}&\mathbf{a}^{\infty}_{out}(x,y,z,t) \\ &= \mathbf{a}^{\infty}_{out}(x,y,z-1,t-1) \qquad \text{by (4)} \\ &= \mathbf{a}(x,y,z-1,t-1) \\ &\quad \text{by induction hypothesis since } \phi_r(x,y,z-1,t-1) = \phi_r(x,y,z,t) \\ &= \mathbf{a}_0(x+\max((t-1)-1-(z-1),0), y+\max((t-1)-1-(z-1),0), \\ &\qquad \max((z-1)-(t-1-1),0),0) \qquad \text{by (10)} \\ &= \mathbf{a}_0(x+\max(t-1-z,0), y+\max(t-1-z,0), \max(z-(t-1),0),0) \\ &= \mathbf{a}(x,y,z,t)\end{aligned}$$

Next we show that $\mathbf{c}^{\infty}_{out} = \mathbf{c}$. If not $\varphi_c \vee \varphi_{c'}$ then $\mathbf{c}^{\infty}_{out} = \perp = \mathbf{c}$. If $\varphi_c \vee \varphi_{c'}$ then

(i) $t = 1$. In this case, $S_1 = S_2 = 0$ since $K_1 = 1$ and $K_2 < 0$.

$$\begin{aligned}\mathbf{c}^{\infty}_{out}(x,y,z,1) &= \begin{cases}\varphi_{h'} \rightarrow \mathbf{c}_0(x,y,z) \\ \varphi_h \rightarrow \mathbf{c}_0(x,y,z) + \mathbf{a}_0(x,y,z) \times \mathbf{b}_0(x,y,z)\end{cases} \qquad \text{by (4)} \\ &= \mathbf{c}(x,y,z,1) + 0 + 0 \qquad \text{by (10)}\end{aligned}$$

(ii) $(t > 1) \wedge \varphi_{h'}$.

$$
\begin{aligned}
&\mathbf{c}^{\infty}_{out}(x,y,z,t)\\
&\quad=\mathbf{c}^{\infty}_{out}(x,y+1,z+1,t-1) \qquad \text{by (4)}\\
&\quad=\mathbf{c}(x,y+1,z+1,t-1)\\
&\qquad \text{by induction hypthesis since } \phi_i(x,y+1,z+1,t-1)=\phi_i(x,y,z,t)\\
&\quad=\mathbf{c}_0(0,(y+1)+(t-1)-1,(z+1)+(t-1)-1,0)+0+0 \qquad \text{by (10)}\\
&\qquad \text{since } \max(x-(t-1),0)=0 \text{ and } K_1=0, K_2<0\\
&\quad=\mathbf{c}_0(0,y+t-1,z+t-1,0)+0+0\\
&\quad=\mathbf{c}(x,y,z,t)
\end{aligned}
$$

(iii) $(t > 1) \wedge \varphi_h \wedge \varphi_c$.

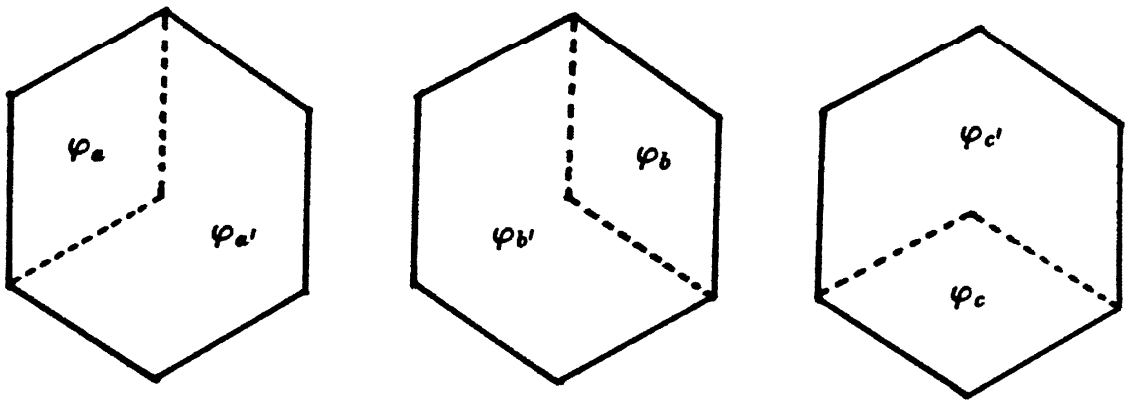
$$
\begin{aligned}
&\mathbf{c}^{\infty}_{out}(0,y,z,t)\\
&=\begin{cases}
\varphi_a \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1)\\
\qquad\qquad +\mathbf{a}_{out}(x+1,y+1,z,t-1)\times\mathbf{b}_{out}(x,y-1,z,t-1)\\
\varphi_b \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1)\\
\qquad\qquad +\mathbf{a}_{out}(x,y,z-1,t-1)\times\mathbf{b}_{out}(x+1,y,z+1,t-1)\\
else \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t-1)\\
\qquad\qquad +\mathbf{a}_{out}(x,y,z-1,t-1)\times\mathbf{b}_{out}(x,y-1,z,t-1)
\end{cases} \qquad \text{by (4)}\\
&=\mathbf{c}_0(0,(y+1)+(t-1-1),(z+1)+(t-1-1),0)\\
&\qquad +\lambda(x,y,z,t).S_1(0,y+1,z+1,t-1)\\
&\qquad +\lambda(x,y,z,t).S_2(0,y+1,z+1,t-1)\\
&\qquad +\mathbf{a}_0(\max(t-1-z,0),y+\max(t-1-z,0),\max(z-(t-1),0),0)\\
&\qquad\quad \times\mathbf{b}_0(\max(t-1-y,0),\max(y-(t-1),0),z+\max(t-1-y,0),0)\\
&\qquad \text{by (10) and the hypothesis (using both the reflected and incident waves)}\\
&=\mathbf{c}_0(0,y+t-1,z+t-1,0)+\lambda(x,y,z,t).S_1(0,y,z,t)\\
&\qquad +\lambda(x,y,z,t).S_2(0,y,z,t)\\
&\qquad \text{since } \lambda(x,y,z,t).S_1(0,y+1,z+1,t-1)=\lambda(x,y,z,t).S_1(0,y,z,t)=0
\end{aligned}
$$

(iv) $(t > 1) \wedge \varphi_h \wedge \varphi_{c'}$.

$$
\begin{aligned}
&\mathbf{c}^{\infty}_{out}(x,y,z,t)\\
&=\begin{cases}\varphi_a \rightarrow \mathbf{c}_{out}(x-1,y,z,t-1)\\
\qquad\quad +\mathbf{a}_{out}(x+1,y+1,z,t-1)\times\mathbf{b}_{out}(x,y-1,z,t-1)\\
\varphi_b \rightarrow \mathbf{c}_{out}(x-1,y,z,t-1)\\
\qquad\quad +\mathbf{a}_{out}(x,y,z-1,t-1)\times\mathbf{b}_{out}(x+1,y,z+1,t-1)\end{cases}\qquad \text{by (4)}\\
&=\mathbf{c}_0(\max(x-1-(t-1-1),0),y+\max(t-1-1-(x-1),0),\\
&\qquad z+\max(t-1-1-(x-1),0),0)\\
&\quad +\lambda(x,y,z,t).S_1(x-1,y,z,t-1)+\lambda(x,y,z,t).S_2(x-1,y,z,t-1)\\
&\quad +\mathbf{a}_0(\max(x+t-1-z,0),y+\max(x+t-1-z,0),\\
&\qquad\qquad \max(z-(t-1)-x,0),0)\\
&\qquad \times\mathbf{b}_0(\max(x+t-1-y,0),\max(y-(t-1)-x,0),\\
&\qquad\qquad z+\max(x+t-1-y,0),0)\\
&\quad \text{by induction hypthesis and (10)}\\
&=\mathbf{c}_0(\max(x-(t-1),0),y+\max(t-1-x,0),z+\max(t-1-x,0),0)\\
&\quad +\lambda(x,y,z,t).S_1(x,y,z,t)+\lambda(x,y,z,t).S_2(x,y,z,t)\\
&\quad \text{since } \lambda(x,y,z,t).S_2(x-1,y,z,t-1)=\lambda(x,y,z,t).S_2(x,y,z,t). \quad \square
\end{aligned}
$$

3. A Self-timed System

A self-timed system has no global clock which synchronizes each invocation of all processes. The time coordinates of processes do not have a simple relationship as a synchronous system does. The initialization is also different. The following expressions indicate the parts of the space domain where elements of matrices are placed initially.



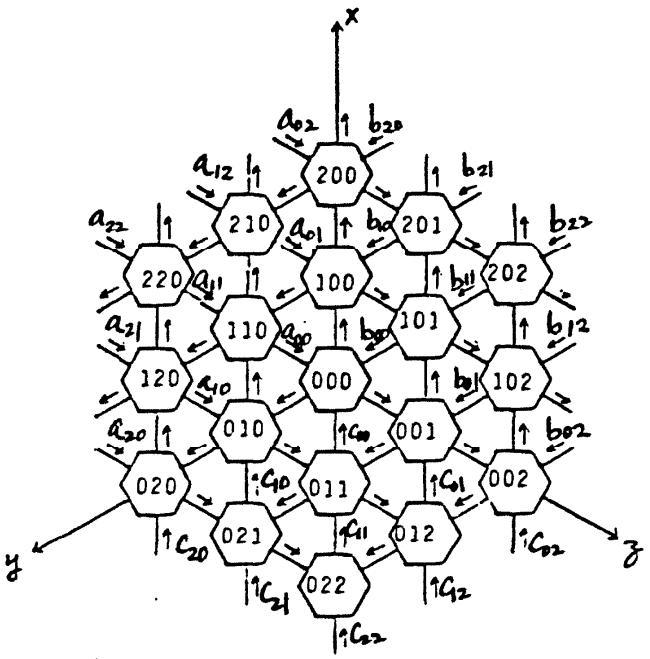


Figure 3–3 The initialization of a self-timed systolic array

$$\varphi_a \equiv \varphi_{xy} \vee \varphi_x \vee \varphi_y$$
$$\varphi_b \equiv \varphi_{xz} \vee \varphi_z \vee \varphi_x$$
$$\varphi_c \equiv \varphi_{yz} \vee \varphi_y \vee \varphi_z$$

The space-time domain:

$$\varphi_s \equiv \varphi_a \vee \varphi_b \vee \varphi_c$$
$$\varphi_t(x,y,z) \equiv 0 < t \leq n - \max(x,y,z)$$

⟨PTERM⟩'s that define the processes:

$$\mathbf{a}_{out}(x,y,z,t_{(x,y,z)}) = \begin{cases} \varphi_h \rightarrow \mathbf{a}_{in}(x,y,z,t_{(x,y,z)}) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{b}_{out}(x,y,z,t_{(x,y,z)}) = \begin{cases} \varphi_h \rightarrow \mathbf{b}_{in}(x,y,z,t_{(x,y,z)}) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{c}_{out}(x,y,z,t_{(x,y,z)}) = \begin{cases} \varphi_h \rightarrow \mathbf{c}_{in}(x,y,z,t_{(x,y,z)}) \\ \qquad +\mathbf{a}_{in}(x,y,z,t_{(x,y,z)}) \times \mathbf{b}_{in}(x,y,z,t_{(x,y,z)}) \\ else \rightarrow \perp \end{cases} \quad (12)$$

⟨CTERM⟩'s that define the connections:

$$\mathbf{a}_{in}(x,y,z,t_{(x,y,z)}) = \begin{cases} t_{(x,y,z)} = 1 \rightarrow \begin{cases} \varphi_a \rightarrow \mathbf{a}_0(x,y,z) \\ \varphi_{a'} \rightarrow \mathbf{a}_{out}(x,y,z-1,t_{(x,y,z-1)}) \\ else \rightarrow \perp \end{cases} \\ t_{(x,y,z)} > 1 \rightarrow \begin{cases} \varphi_a \rightarrow \mathbf{a}_{out}(x+1,y+1,z,t_{(x+1,y+1,z)}) \\ \varphi_{a'} \rightarrow \mathbf{a}_{out}(x,y,z-1,t_{(x,y,z-1)}) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$\mathbf{b}_{in}(x,y,z,t_{(x,y,z)}) = \begin{cases} t_{(x,y,z)} = 1 \rightarrow \begin{cases} \varphi_b \rightarrow \mathbf{b}_0(x,y,z) \\ \varphi_{b'} \rightarrow \mathbf{b}_{out}(x,y-1,z,t_{(x,y-1,z)}) \\ else \rightarrow \perp \end{cases} \\ t_{(x,y,z)} > 1 \rightarrow \begin{cases} \varphi_b \rightarrow \mathbf{b}_{out}(x+1,y,z+1,t_{(x+1,y,z+1)}) \\ \varphi_b \rightarrow \mathbf{b}_{out}(x,y-1,z,t_{(x,y-1,z)}) \\ else \rightarrow \perp \end{cases} \end{cases}$$

$$\mathbf{c}_{in}(x,y,z,t_{(x,y,z)}) = \begin{cases} t_{(x,y,z)} = 1 \rightarrow \begin{cases} \varphi_c \rightarrow \mathbf{c}_0(x,y,z) \\ \varphi_{c'} \rightarrow \mathbf{c}_{out}(x-1,y,z,t_{(x-1,y,z)}) \\ else \rightarrow \perp \end{cases} \\ t_{(x,y,z)} > 1 \rightarrow \begin{cases} \varphi_c \rightarrow \mathbf{c}_{out}(x,y+1,z+1,t_{(x,y+1,z+1)}) \\ \varphi_{c'} \rightarrow \mathbf{c}_{out}(x-1,y,z,t_{(x-1,y,z)}) \\ else \rightarrow \perp \end{cases} \end{cases} \tag{13}$$

Note that there are free variables in the expressions of ⟨CTERM⟩'s. They are the time coordinates of neighboring processes which communicate with process (x,y,z). To obtain the semantics of the algorithm, the semantics of these free variables must be obtained first. Since the communications between neighboring processes determine the relationships of the time coordinates of the processes, the time coordinate of one can be expressed as a function of

the other. The correctness of these functions that relate the neighboring processes must be proven. Since a communication is established by input ports being filled by neighboring processes or by initialization of the system, such a function depends on the initial input functions $\mathbf{a}_0$, $\mathbf{b}_0$, and $\mathbf{c}_0$. They are defined as follows.

Input Mapping Functions. The functions (I_a, J_a), (I_b, J_b) and (I_c, J_c) which map from the space coordinates to index pair of matrices are elements of $[\mathcal{E} \rightarrow \mathcal{N}^2]$.

$$I_a(x,y,z) \equiv \begin{cases} \varphi_a \rightarrow y \\ else \rightarrow \perp \end{cases}$$

$$J_a(x,y,z) \equiv \begin{cases} \varphi_a \rightarrow x \\ else \rightarrow \perp \end{cases}$$

$$I_b(x,y,z) \equiv \begin{cases} \varphi_b \rightarrow x \\ else \rightarrow \perp \end{cases}$$

$$J_b(x,y,z) \equiv \begin{cases} \varphi_b \rightarrow z \\ else \rightarrow \perp \end{cases}$$

$$I_c(x,y,z) \equiv \begin{cases} \varphi_c \rightarrow y \\ else \rightarrow \perp \end{cases}$$

$$J_c(x,y,z) \equiv \begin{cases} \varphi_c \rightarrow z \\ else \rightarrow \perp \end{cases}$$

The initial input functions $\mathbf{a}_0$, $\mathbf{b}_0$, and $\mathbf{c}_0$,.

$$\mathbf{a}_0(x,y,z) \equiv \begin{cases} \varphi_a \rightarrow A(I_a(x,y,z), J_a(x,y,z)) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{b}_0(x,y,z) \equiv \begin{cases} \varphi_b \rightarrow A(I_b(x,y,z), J_b(x,y,z)) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{c}_0(x,y,z) \equiv \begin{cases} \varphi_c \rightarrow A(I_c(x,y,z), J_c(x,y,z)) \\ else \rightarrow \perp \end{cases} \qquad (14)$$

Let $s \equiv (x, y, z)$, $t(s) \equiv t_{(x,y,z)}$, and $w(s) \equiv x + y + z$. In order to relate time coordinates of processes, we look at two processes s and s' where an output port p of s' is connected to an input port of s and the following condition holds:

> for each invocation, p is always selected by process s as one of the input ports and always selected by s' as one of its output ports.

We claim that

Proposition 3:
Whenever output port p is filled by $t(s')$'th invocation of s' and process s is ready (all of its input ports are filled, see step (ix), Section 1 of Chapter 2) for its $t(s)$'th invocation then

$$t(s) - t(s') = c \tag{15}$$

holds for some constant integer c.

Proof :

Let equation (15) hold initially for some c, $t(s) = t_0$, and $t(s') = t'_0$. We show that (15) still holds if process s is ready for a new invocation $t_0 + 1$, i.e., to show that p is filled by $t'_0 + 1$'th invocation of s'. If process s is ready for $t_0 + 1$'th invocation, then p must have been filled again since s always selects p and the last item in p is taken by t_0'th invocation of s. Hence process s' must have completed at least one more invocation and filled its output port p, i.e., $t(s') \geq t'_0 + 1$.

On the other hand, the number of times s' can be invoked is contrained by s. Initially process s is ready to start its t_0'th invocation. Process s' can start its $t'_0 + 1$'th invocation if it is ready. Since p is always one of the selected output ports of s', s' cannot complete its $t'_0 + 1$ invocation

(hence starts its $s' + 2$'th invocation) without s having started its t_0'th invocation and emptied p which was filled by the t_0''th invocation of s'. Therefore, p is filled by $t(s')$'th invocation of s' where $t(s') \leq t_0' + 1$. This proves that (15) holds. □

For this algorithm, the following relationships among invocations of neighboring processes holds.

$$t(s') = \begin{cases} w(s') = w(s) + 2 \rightarrow t(s) - 1 \\ w(s') = w(s) - 1 \rightarrow t(s) \end{cases} \tag{16}$$

We can use Proposition 3 because in this algorithm all processes always select all input ports and all output ports. If process s is closer to the origin than s', i.e., $w(s') = w(s) + 2$, then by (14), p is filled at the initialization. Hence $c = 1$ in this case. If process s is further away from the origin than s', i.e., $w(s') = w(s) - 1$, then by (14), p is not filled at the initialization and therefore $c = 0$.

By replacing the free variables $t_{(x',y',z')}$ in (13) by a function of $t_{(x,y,z)}(\equiv t(s))$ defined by (16), all variables in (13) become bound. A set of equations similar to (4) can be obtained and the functionality of this self-timed systolic algorithm can be verified in the same way as the synchronous case. Before verifying the functionality of the algorithm, we need to show that the algorithm is free of deadlock, i.e., for each process (x, y, z), all $k = n - \max(x, y, z)$ invocations required for computing the result (defined by the time domain φ_t) are realizable. By initialization, process $(0, 0, 0)$ can start its first invocation since all of its three inputs are filled. However, the number of invocations is also constrained by the initialization, since any process (x, y, z) where $k = 1$, i.e., on the boundary of the array, can be invoked at most one time only. It is easy to show that these boundary processes will be invoked at least once by induction on $w = x + y + z$. The first invocation of all processes thus

results in a configuration of the ports being filled exactly the same as the initialization except for the boundary ones. Hence by induction on k we can show that any process (x, y, z) will be invoked k times. □

Now we can assert that the least fixed-point of the algorithm is the following:

Proposition 4:

$$\mathbf{a}^{\infty}_{out}(x,y,z,t) \equiv \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow \mathbf{a}_0(x+t, y+t, 0) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{b}^{\infty}_{out}(x,y,z,t) \equiv \begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow \mathbf{b}_0(x+t, 0, z+t) \\ else \rightarrow \perp \end{cases}$$

$$\mathbf{c}^{\infty}_{out}(x,y,z,t) = \begin{cases} \varphi_c \vee \varphi_{c'} \rightarrow \mathbf{c}_0(\max(x-t,0), y+\max(t-x,0), z+\max(t-x,0)) \\ \qquad + \sum_{k=0}^{x+t} \mathbf{a}_0(k, y+t, 0) \times \mathbf{b}_{in}(k, 0, z+t) \\ else \rightarrow \perp \end{cases} \tag{17}$$

Proof :

Similar to that of Proposition 2, the proof is by induction. In this case, the induction is over the well-founded set $K \equiv \{ k : k \equiv x+y+z+3(t_{(x,y,z)}) \}$ for each phase of the wave. The phase of the reflected wave is $\phi_r = t_{(x,y,z)}$ and that of the incident wave is $\phi_i = x+y+z+2t_{(x,y,z)}$. □.

Similar to the synchronous case, using the following set of output mapping functions in (9), Proposition 1 holds for the self-timed system.

Output Mapping Functions. (X_a, Y_a, Z_a, T_a), (X_b, Y_b, Z_b, T_b), and (X_c, Y_c, Z_c, T_c) where

$X_a(i,j) \equiv j - \min(i,j), Y_a(i,j) \equiv i - \min(i,j), Z_a(i,j) \equiv n-1-\min(i,j).$
$X_b(i,j) \equiv i - \min(i,j), Y_b(i,j) \equiv n-1-\min(i,j), Z_b(i,j) \equiv j - \min(i,j).$
$X_c(i,j) \equiv n-1-\min(i,j), Y_c(i,j) \equiv i - \min(i,j), Z_c(i,j) \equiv j - \min(i,j).$
$T_a(i,j) \equiv T_b(i,j) \equiv T_c(i,j) \equiv \min(i,j).$ (18)

From both algorithms, we observe that the input and output mapping functions and the behavior description of the hexagonal array are much simpler for the self-timed version. This result is not accidental, for the interaction among flows of data for this particular algorithm only utilizes one third of the resources (time and space). In the self-timed version, only one third of the processes (all processes with the same $k \equiv x+y+z+3t_{(x,y,z)}$) are active at any instant. In the synchronous version, all processes are active at all times; thus padding zeros are necessary since only one third of the inputs are "real" data. The simplicity of the self-timed version is a pay-off of the more sophisticated synchronization method. It is necessary to prove that local synchronization gives rise to a relationship of the time coordinates among all the neighboring processes and the computation is deadlock free.

In summary, the behavior of a systolic array is obtained by:

(i) An input mapping function from the structure of the value domain to the space-time structure of the system.

(ii) The fixed-point of the space-time algorithm which defines the computation of the system in the space-time domain.

(iii) An output mapping function from the space-time structure to the structure of the value domain.

4. Pipelined Architecture

The pipelined architecture (Figure 1–2) is very similar to the systolic architecture (Figure 1–1) in that local communication is used to avoid long propagation delay. It is usually simpler to describe and analyze because of the one dimensional structure in space. In the example presented below,

each stage of the pipeline has an internal state, which is not the case in the systolic array example. In CRYSTAL, since the time coordinate is explicitly used, internal state pose no difficulty in describing a system.

The following is the space-time algorithm for an n-stage pipelined inner product element IPE given the behavioral description of a one bit inner product element IPB shown in Figure 3–4. The function this pipeline implements is $B \times M + A_{in}$ where B and M are n-bit non-negative binary numbers and A_{in} is a $2n$-bit non-negative binary number.

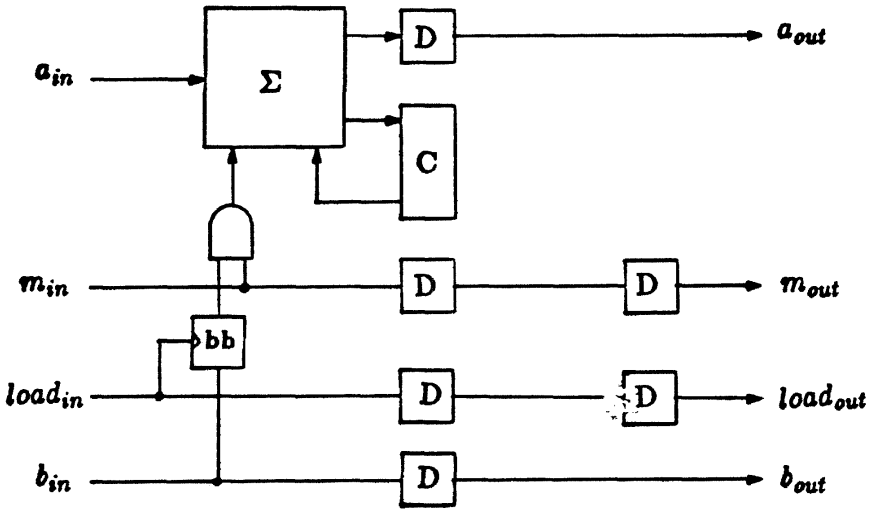


Figure 3–4 A single bit inner product element IPB

We show that the behavior of an IPE element (the composition of n IPB elements and several n bit shift registers) is in fact the function above. The symbol "$\cdot$", and "+" denotes the "and" and "or" operations on boolean values $\hat{0}$ and $\hat{1}$. Let $S \equiv \{0, 1, 2, \ldots, n\}$ be the space domain for indexing the IPB elements and shift registers and $\mathcal{T} \equiv \{0, 1, 2, \ldots\}$ be the set of non-negative integers which indicates the steps of computation. Since the behavior of this

pipeline inner product element is periodic in time, the value of the expression $t-s$ can be restricted to within 1 and $2n$. The space-time domain $\mathcal{E}$ for the pipeline is defined as

$$\mathcal{E} \equiv \{(s,t) : s \in S, t \in T, \text{ and } 1 \leq t-s \leq 2n\}$$

The value of $\mathbf{a}(s,t)$ is the accumulated partial sum computed by IPB element s at step t. $\mathbf{c}(s,t)$ and $\mathbf{bb}(s,t)$ are the carry of an adder and the s'th bit of multiplier B, respectively, both are internal states of IPB element s. Signal $\mathbf{load}(s,t)$ is for loading $\mathbf{bb}(s,t)$ from $\mathbf{b}(s,t)$, a bit of the multiplier. Signal $\mathbf{k}(s,t)$ is used to clear the internal state of the carry and multiplier of the previous word before a new word starts. The value of $\mathbf{k}(s,t)$ computed at the $k-1$'th cycle (word) is used as an initial condition for the k'th word. In this case, t and s such that $t-s=0$ which is not in the specified space-time domain is used in place of t and s such that $t-s=2n$ in the description of $\mathbf{k}$.

$$\mathbf{a} \in [\mathcal{E} \to B]$$
$$\mathbf{a}(s,t) = \begin{cases} s > 0 \to \mathbf{a}(s-1,t-1) \oplus (\mathbf{bb}(s,t) \cdot \mathbf{m}(s-1,t-2)) \oplus \mathbf{c}(s,t-1) \\ s = 0 \to \mathrm{A}_0(t) \end{cases} \tag{1}$$

$$\mathbf{c} \in [\mathcal{E} \to B]$$
$$\mathbf{c}(s,t) = \begin{cases} s > 0 \to \\ \quad \mathbf{MAJ}(\mathbf{a}(s-1,t-1), (\mathbf{bb}(s,t) \cdot \mathbf{m}(s-1,t-2)), \\ \qquad \mathbf{c}(s,t-1)) \cdot \overline{\mathbf{k}}(s-1,t-1) + \hat{0} \cdot \mathbf{k}(s-1,t-1) \\ s = 0 \to \perp \end{cases} \tag{2}$$

$$\mathbf{bb} =\in [\mathcal{E} \to B]$$
$$\mathbf{bb}(s,t)\begin{cases} s > 0 \to (\mathbf{bb}(s,t-1) \cdot \overline{\mathbf{load}}(s-1,t-2) \\ \qquad +\mathbf{b}(s-1,t-1) \cdot \mathbf{load}(s-1,t-2)) \cdot \overline{\mathbf{k}}(s-1,t-1) \\ \qquad +\hat{0} \cdot \mathbf{k}(s-1,t-1) \\ s = 0 \to \perp \end{cases} \tag{3}$$

The following are shift registers (delay elements) for propagating the operands. Two register cells (two units of delay) are used for each of the

signals **load** and **m**, whereas one register cell is used for each of the signals **k** and **b** in an IPB element.

$$\mathbf{load} \in [\mathcal{E} \to B]$$

$$\mathbf{load}(s,t) = \begin{cases} (s > 0) \wedge (t \geq 2) \to \mathbf{load}(s-1, t-2) \\ s = 0 \to \begin{cases} t = 0 \to \hat{1} \\ t > 0 \to \hat{0} \end{cases} \end{cases}$$

therefore

$$\mathbf{load}(s,t) = \begin{cases} t = 2s \to \hat{1} \\ s < t < 2s \to \perp \\ 2s < t \leq 2n + s \to \hat{0} \end{cases} \tag{4}$$

$$\mathbf{k} \in [\mathcal{E} \to B]$$

$$\mathbf{k}(s,t) = \begin{cases} (s > 0) \wedge (t \geq 1) \to \mathbf{k}(s-1, t-1) \\ s = 0 \to \begin{cases} t = 0 \to \hat{1} \\ 0 < t \leq 2n - 1 \to \hat{0} \end{cases} \end{cases}$$

therefore

$$\mathbf{k}(s,t) = \begin{cases} t = s \to \hat{1} \\ s < t \leq 2n + s - 1 \to \hat{0} \end{cases} \tag{5}$$

$$\mathbf{b} \in [\mathcal{E} \to B]$$

$$\mathbf{b}(s,t) = \begin{cases} (s > 0) \wedge (t \geq 1) \to \mathbf{b}(s-1, t-1) \\ s = 0 \to \begin{cases} 0 < t \leq n \to \mathrm{B}(t-1) \\ n < t \leq 2n \to \perp \end{cases} \end{cases}$$

therefore

$$\mathbf{b}(s,t) = \begin{cases} s < t \leq n + s \to \mathrm{B}(t-1-s) \\ n + s < t \leq 2n + s \to \perp \end{cases} \tag{6}$$

$$\mathbf{m} \in [\mathcal{E} \to B]$$

$$\mathbf{m}(s,t) = \begin{cases} (s > 0) \wedge (t \geq 2) \to \mathbf{m}(s-1, t-2) \\ s = 0 \to \begin{cases} 0 \leq t < n \to \mathrm{M}(t) \\ n \leq t < 2n \to \hat{0} \end{cases} \end{cases}$$

therefore

$$\mathbf{m}(s,t) = \begin{cases} s < t < 2s \to \perp \\ 2s \leq t < n + 2s \to \mathrm{M}(t-2s) \\ (n + 2s \leq t \leq 2n + s) \to \hat{0} \end{cases} \tag{7}$$

From the functional description of shift registers, (4), (5) and (6), the functional description of the internal state **bb** can be derived.

$$\mathbf{bb}(s,t) = \begin{cases} t = s \to \hat{0} \\ t = 2s \to \mathbf{b}(s-1, t-1) = \mathrm{B}(t-1-s) = \mathrm{B}(s-1) \\ (s < t \leq 2s-1) \vee (2s < t \leq 2n+s-1) \to \mathbf{bb}(s, t-1) \\ t = 2n+s \to \hat{0} \end{cases} \tag{8}$$

$$= \begin{cases} s < t \leq 2s-1 \to \hat{0} \\ 2s \leq t \leq 2n+s-1 \to \mathrm{B}(s-1) \\ t = 2n+s \to \hat{0} \end{cases}$$

To prove the correctness of this algorithm, the space-time structure is first mapped to another structure. The multiplication of two n-bit binary numbers can be represented by a recurrence of the partial product. In the following i is used to indicate the partial product at stage i and j is used to indicate bit j. Let

$$N_1 \equiv \{0, 1, 2, \ldots, n\} \text{ and } N_2 \equiv \{0, 1, 2, \ldots, 2n-1\},$$

and A and C be two functions in $[N_1 \times N_2 \to B]$ where

$$\mathrm{A}(i,j) \equiv \mathbf{a}(i, i+j+1) \text{ and } \mathrm{C}(i,j) \equiv \mathbf{c}(i, i+j+1).$$

Inversely,

$$\mathbf{a}(s,t) \equiv \mathrm{A}(s, t-1-s) \text{ and } \mathbf{c}(s,t) \equiv \mathrm{C}(s, t-1-s) \tag{9}$$

Using definition (9) and substituting the result obtained in (7) and (8) into (1) and (2), we obtain the following relations among the new functions A, C, B and M.

$$\mathbf{a}(s,t) = \mathrm{A}(s, t-1-s)$$
$$= \begin{cases} s > 0 \to \\ \quad \begin{cases} 2s \le t \le n+2s-1 \to \\ \quad \mathrm{A}(s-1,(t-1)-1-(s-1)) \oplus (\mathrm{B}(s-1)\cdot \mathrm{M}((t-2)-2(s-1))) \\ \quad \oplus \mathrm{C}(s,(t-1)-s) \\ (s < t \le 2s-1) \vee (n+2s-1 < t \le 2n+s) \to \\ \quad \mathrm{A}(s-1,(t-1)-1-(s-1)) \oplus \mathrm{C}(s,(t-1)-1-s) \end{cases} \\ s = 0 \to \mathrm{A}_0(t) \end{cases} \tag{10}$$

$$\mathbf{c}(s,t) = \mathrm{C}(s, t-1-s)$$
$$= \begin{cases} s > 0 \to \\ \quad \begin{cases} 2s \le t \le n+2s-1 \to \\ \quad \mathbf{MAJ}(\mathrm{A}(s-1,(t-1)-1-(s-1)), \\ \quad (\mathrm{B}(s-1)\cdot \mathrm{M}((t-2)-2(s-1))), \mathrm{C}(s,(t-1)-1-s)) \\ (s < t \le 2s-1) \vee (n+2s-1 < t \le 2n+s) \to \\ \quad \mathbf{MAJ}(\mathrm{A}(s-1,(t-1)-1-(s-1)), 0, \mathrm{C}(s,(t-1)-1-s)) \\ t = 2n+s-1 \to \hat{0} \end{cases} \\ s = 0 \to \bot \end{cases} \tag{11}$$

Let $i = s$ and $j = t-1-s$, substituting them into (9) and (10) we obtain

$$\mathrm{A}(i,j) = \begin{cases} i > 0 \to \begin{cases} i-1 \le j < n+i-1 \to \\ \quad \mathrm{A}(i-1,j) \oplus (\mathrm{B}(i-1)\cdot \mathrm{M}(j-(i-1))) \oplus \mathrm{C}(i,j-1) \\ (0 \le j < i-1) \vee (n+i-1 \le j < 2n) \to \\ \quad \mathrm{A}(i-1,j) \oplus \mathrm{C}(i,j-1) \end{cases} \\ i = 0 \to \mathrm{A}_0(j) \end{cases} \tag{12}$$

$$\mathrm{C}(i,j) = \begin{cases} i > 0 \to \begin{cases} i-1 \le j < n+i-1 \to \\ \quad \mathbf{MAJ}(\mathrm{A}(i-1,j), (\mathrm{B}(i-1)\cdot \mathrm{M}(j-(i-1))), \mathrm{C}(i,j-1)) \\ (0 \le j < i-1) \vee (n+i-1 \le j < 2n-1) \to \\ \quad \mathbf{MAJ}(\mathrm{A}(i-1,j), \hat{0}, \mathrm{C}(i,j-1)) \\ j = 2n-1 \to \hat{0} \end{cases} \\ i = 0 \to \bot \end{cases} \tag{13}$$

In the above two equations, if j is not in $\mathcal{N}_2$, it is understood as $2n + j$ of the previous word computed in the pipeline. We now proceed to show that equation (12) and (13) embody an algorithm for computing the inner product of positive numbers in the binary representation.

Let $\mathbf{x}(k)$, $\mathbf{y}(k)$, and $\mathbf{z}(k)$ for $k = 0, 1, 2, \ldots, n-1$ be the binary representation of non-negative integers x, y and z, respectively. A recurrence formula for addition of binary numbers $z = x + y$ is

$$\begin{aligned} \mathbf{z}(k) &= \mathbf{y}(k) \oplus \mathbf{x}(k) \oplus \mathbf{w}(k-1) \\ \mathbf{w}(k) &= \mathrm{MAJ}(\mathbf{y}(k), \mathbf{x}(k), \mathbf{w}(k-1)) \\ &\quad \text{for } k = 0, 1, \ldots, n-1 \text{ and } \mathbf{z}(n) = \mathbf{w}(n-1) \\ &\quad \text{where } \mathbf{w}(k)\text{'s are the carry bits and} \\ &\quad \mathbf{w}(-1) = \hat{0}. \end{aligned} \tag{14}$$

Notice that the boolean *exclusive or* function is actually modulo 2 addition of bits and the *majority* function is the floor ("integer" division) of sum of bits divided by 2, i.e.,

$$\begin{aligned} \mathbf{z}(k) &= (\mathbf{y}(k) + \mathbf{x}(k) + \mathbf{w}(k-1)) \pmod 2 \\ \mathbf{w}(k) &= \left\lfloor \frac{(\mathbf{y}(k) + \mathbf{x}(k) + \mathbf{w}(k-1))}{2} \right\rfloor \end{aligned} \tag{15}$$

By a straight forward induction on n, it can be shown that

$$\sum_{k=0}^{n} 2^k \mathbf{z}(k) = \sum_{k=0}^{n-1} 2^k \mathbf{x}(k) + \sum_{k=0}^{n-1} 2^k \mathbf{y}(k).$$

Due to the finiteness of machines, usually $z = x + y \pmod{2^n}$ is the operation performed. This operation is defined similarly by

$$\begin{aligned} \mathbf{z}(k) &= \mathbf{y}(k) \oplus \mathbf{x}(k) \oplus \mathbf{w}(k-1) \\ &\quad \text{for } k = 0, 1, \ldots, n-1 \text{ and} \\ \mathbf{w}(k) &= \mathrm{MAJ}(\mathbf{y}(k), \mathbf{x}(k), \mathbf{w}(k-1)) \\ &\quad \text{for } k = 0, 1, \ldots, n-2 \text{ and } \mathbf{w}(-1) = \hat{0} \\ &\quad \text{and } \mathbf{z}(n) = \mathbf{w}(n-1) = \hat{0} \end{aligned} \tag{16}$$

For the pipeline inner product element, we need to show that

$$\sum_{j=0}^{2n-1} 2^j \mathrm{A}(n,j) = \left(\sum_{k=0}^{n-1} 2^k \mathrm{B}(k) \times \sum_{k=0}^{n-1} 2^k \mathrm{M}(k) + \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) \right) (\mathrm{mod}\ 2^{2n})$$

Since

$$\begin{aligned}
&\sum_{k=0}^{n-1} 2^k \mathrm{B}(k) \times \sum_{k=0}^{n-1} 2^k \mathrm{M}(k) + \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) \\
&= \sum_{i=0}^{n-1} \left(2^i \mathrm{B}(i) \times \sum_{k=0}^{n-1} 2^k \mathrm{M}(k) \right) + \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) \\
&= \sum_{i=0}^{n-1} d_i + \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j)
\end{aligned}$$

$$\text{where } d_i = 2^i \mathrm{B}(i) \times \sum_{k=0}^{n-1} 2^k \mathrm{M}(k) = \sum_{j=i}^{n+i-1} 2^j \mathrm{M}(j-i)\mathrm{B}(i)$$

$$= \left(\left(\cdots \left(\left(\sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) + d_0 \right) + d_1 \right) + \cdots \right) + d_{n-1} \right)$$

$$= p_n \text{ (the } n\text{'th partial product)}$$

$$\text{where } \begin{cases} p_i = p_{i-1} + d_{i-1}, & i = 1, 2, \ldots, n \\ p_0 = \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) \end{cases}$$

By induction on i and (16), it can easily be shown that

$$p_i\ (\mathrm{mod}\ 2^{2n}) = \sum_{j=0}^{2n-1} 2^j \mathrm{A}(i,j) \text{ and therefore}$$

$$\begin{aligned}
\sum_{j=0}^{2n-1} 2^j \mathrm{A}(n,j) &= p_n\ (\mathrm{mod}\ 2^{2n}) \\
&= \left(\sum_{k=0}^{n-1} 2^k \mathrm{B}(k) \times \sum_{k=0}^{n-1} 2^k \mathrm{M}(k) + \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j) \right) (\mathrm{mod}\ 2^{2n})
\end{aligned}$$

Thus the correctness of the above pipelined algorithm has been shown. This proof is approached differently from those of the systolic arrays in the previous sections where the solution with space-time domain is obtained and

then mapped to the structure of matrices. In this proof, the space-time coordinates are mapped to the coordinates indicating the stage of the partial product and the bit number for the equation. The solution is therefore obtained with these indices as domains. It is because these indices are more convenient for expressing the multiplication function as sequences of bits. We now let $A_{out} = \sum_{j=0}^{2n-1} 2^j \mathrm{A}(n,j)$, $A_{in} = \sum_{j=0}^{2n-1} 2^j \mathrm{A}_0(j)$, $B = \sum_{k=0}^{n-1} 2^k \mathrm{B}(k)$, and $M = \sum_{k=0}^{n-1} 2^k \mathrm{M}(k)$. The functionality of the pipeline can be described by $A_{out} = A_{in} + B \times M$. The data type is a bounded integer rather than a bit. The IPE element can thus be used at the next level without its implementation detail but only its functionality.

Chapter 4

Transistor Networks

Examples of primitive state transition functions given in the previous Chapter are familiar mathematically defined functions. To describe a transistor network, however, functions which model the circuit components must be given. A satisfactory circuit model must be an abstraction at a suitable level of the detailed physical behavior of circuits. A model must be justified from the underlying electrical model with a given set of assumptions made about the particular technology used for the components. The ingenious part in devising such a model is to attain a set of assumptions which have a high enough level of abstraction to allow a model that is computationally practical but still has enough power to describe real circuits. We have chosen Bryant's [3] switch level model of metal-oxide-semiconductor (MOS) technology as the primitive functions for transistor networks. This model has been widely tested by users of his simulator MOSSIM [4]. A transistor network is approximated by a series of conductance networks, each of which is represented by a system of linear equations in this model. A "unit-delay" timing model, which Bryant defined operationally, is used to obtain a new conductance network from the result of the previous conductance network. This model provides the resulting values of all nodes of a transistor network given initial state and values on input nodes. For the purpose of simulating an entire design at the transistor level, Bryant's model alone suffices. However, hierarchical simulation and verification of a network require obtaining the

behavior (a function) of a each sub-network rather than just the values of nodes of each sub-network. It is therefore necessary to re-formulate the entire switch-level model as a space-time algorithm and thus provide formal semantics for an arbitrary transistor network. The functional abstraction of the network can then be used as a building block at the next level. In this way, a VLSI system can be described and verified hierarchically in a simple and uniform manner all the way from networks of transistors to high level networks of processors.

1. Circuit Components as Processes

A VLSI circuit is composed of primitive circuit components such as transistors and nodes. Each of these components has certain attributes such as voltage, current, capacitance, and resistance associated with it. Each of them performs actions which are governed by physical laws defining relations among the attributes. A component (a transistor or a node) is represented as a process. Attributes such as capacitance and resistance are constants associated with this process. Voltages on capacitors are represented by the state of the process. The inputs and outputs of the process (called *signals*), are used to represent the flow of current. The actions performed by each component are represented by state transition functions. For a network of circuit components as a whole, signals are initially defined only at the sources. They gradually propagate in space and reach more components, influence one another and therefore change with time, until they reach a steady state. That is, the signals at all nodes of a circuit do not change anymore. But for an oscillatory circuit this situation may not occur and there exists no steady state. The dynamics of a circuit can be viewed as initial signals, which are undefined over most of the network, evolving into better and better signals (a chain of monotonically more defined functions). This phenomenon

corresponds to our formalization of a signal as a data stream (an unknown function from the space-time domain to the data type of signal values).

2. Data Types for Transistor Networks

Possible data types for a VLSI system include:

(i) The set of boolean values $\mathcal{B} \equiv \{\hat{0}, \hat{1}\}$.

(ii) The set of ternary logic values $\mathcal{V} \equiv \{\hat{0}, \hat{1}, \mathtt{X}\}$ where X represents an illegal voltage between logic 0 ($\hat{0}$) and logic 1 ($\hat{1}$).

(iii) The set of transistor types $\mathcal{P} \equiv \{dtype, ntype, ptype\}$, which denote n-channel depletion mode, and n-channel or p-channel enhancement mode transistors, respectively.

(iv) $\mathcal{C} \equiv \{c_1, c_2, \ldots, c_\kappa\}$ is a discrete set of capacitance strengths : a subset of the set of positive integers. These strengths are ordered by the usual ordering on integers, namely, $c_1 < c_2 < \cdots < c_\kappa$.

(v) $\mathcal{G} \equiv \{g_1, g_2, \ldots, g_\gamma\}$. Similarly, a discrete set of conductance strengths where $g_1 < g_2 < \cdots < g_\gamma$.

(vi) The set of signal strengths $\mathcal{S} \equiv \{0\} + \mathcal{C} + \mathcal{G}$: a sum of data types. The switch level model assumes the capacitance strength is always weaker than the conductance strength except for c_κ which is the capacitance strength of sources. This set is totally ordered as $0 < c_1 < c_2 < \cdots < c_{\kappa-1} < g_1 < g_2 < \cdots < g_\gamma < c_\kappa$. Each adjacent pair of strengths model actual conductance or capacitance that differ by order of magnitude.

(vii) The set of signal values $\Sigma \equiv \mathcal{S} \times \mathcal{V}$: a product of data types. Each signal value has both a signal strength and a logic value.

(viii) A set of functions from data type $\mathcal{E}$ (space-time) to $\mathcal{S}$ (signal strength): a set of continuous functions over data types.

3. Primitives — MOS Switch Level Model

An MOS transistor has three terminals: source, drain, and gate. The voltage on the gate of such a transistor controls the current flow between the source and drain. The source and drain are completely symmetrical. In [3], the behavior of a transistor network is approximated by a series of *logical conductance networks*. Transistors that are *on* are represented as a high conductance and those that are *off* as a zero conductance. Different sized transistors result in conductances which differ in order of magnitude and different sized nodes have capacitances that differ in order of magnitude. The steady state of each conductance network affects the topology of the next conductance network by the gate voltage which controls the on or off state of each transistor. The transition from one conductance network topology to the next is based on the unit-delay model of switching a transistor with respect to its gate voltage.

For each conductance network, the Thévenin equivalent circuit is first constructed from the original network. For each node, the equivalent admittance with respect to all sources (combining the admittance accumulated along each path to a source or input) is computed. The voltage for each node of the equivalent circuit is then obtained.

In order to detect whether there is any danger of charge sharing or paths to both logic 1 and logic zero in a conductance network, Ternary logic values are used for node voltages. If the gate voltage of a transistor is $x \in \mathcal{V}$, it is ambiguous what the conductance for that transistor is. An effective way of handling this case is by obtaining three separate strengths and then computing the strength and voltage for each node. First the equivalent admittance is computed from all sources and all stored charge of voltage,

both $\hat{1}$ and $\hat{0}$, assuming all transistors with X on their gates are off. Then the equivalent admittances from $\hat{1}$ sources or stored charge and $\hat{0}$ sources or stored charge are computed separately where transistors with x on their gate are treated as on. These admittances are called the one part and zero part, respectively. Once all three admittances are evaluated, they are combined to produce the new node strength and voltage. The following is a hierarchy of processes which models an arbitrary transistor network. For each process, its states and state transition functions are described. In defining the functions, formal parameters are used and should not be confused with the states if the same symbols are used. The inputs and outputs are not explicitly differentiated from the states where a process is defined although this information is in the description at a higher level where the process is used.

4. Transistor as a Process

States.

4.1 A constant $g_0 \in \mathcal{G}$ which is the conductance strength of the transistor when it is on.

4.2 $g_1 \in \mathcal{G}$ and $g_2 \in \mathcal{G}$, the conductances whose values depend on gate voltage of the transistor. Both g_1 and g_2 equal g_0 if the gate voltage is such that the transistor is on and zero if the transistor is off. When the gate voltage is X, g_1 is the *minimum* conductance in the sense that the transistor is treated as if it were off and g_2 is the *maximum* in the sense that the transistor considered as on.

4.3 $p \in P$, a constant specifying the type of the transistor.

4.4 $sr \in S$ and $dr \in S$, signal strengths computed by the transistor at its source and drain.

State Transition Functions

4.5 The function *couple* models a signal strength s coupled with a conductance g. The node strength (or equivalent conductance) is affected by the conductance g connecting (in series) to it, The effect of two conductance in series is treated as the weaker one of the two since the order of magnitude approximation is used.

$$couple \in [\mathcal{S} \times \mathcal{G} \to \mathcal{S}]$$
$$couple(s, g) \equiv \min(s, g)$$

4.6 The functions $setconduc_1$ and $setconduc_2$ model the control of the transistor conductance by the gate voltage v. $setconduc_1$ gives the minimum conductance g_1 and $setconduc_2$ gives the maximum conductance g_2 mentioned above.

$$setconduc_1 \in [\mathcal{V} \times \mathcal{P} \times \mathcal{G} \to \mathcal{G}]$$
$$setconduc_1(v, p, g) \equiv \begin{cases} p = dtype \to g \\ p = ntype \to \begin{cases} (v = \hat{0}) \vee (v = \mathrm{X}) \to 0 \\ v = \hat{1} \to g \end{cases} \\ p = ptype \to \begin{cases} (v = \hat{1}) \vee (v = \mathrm{X}) \to 0 \\ v = \hat{0} \to g \end{cases} \end{cases}$$

$$setconduc_2 \in [\mathcal{V} \times \mathcal{P} \times \mathcal{G} \to \mathcal{G}]$$
$$setconduc_2(v, p, g) \equiv \begin{cases} p = dtype \to g \\ p = ntype \to \begin{cases} v = \hat{0} \to 0 \\ (v = \hat{1}) \vee (v = \mathrm{X}) \to g \end{cases} \\ p = ptype \to \begin{cases} v = \hat{1} \to 0 \\ (v = \hat{0}) \vee (v = \mathrm{X}) \to g \end{cases} \end{cases}$$

5. Node as a Process

States.

5.1 The node capacitance $c \in C$.

5.2 The equivalent admittance of the node $a \in S$.

5.3 The node logic value $v \in \mathcal{V}$ and node strength $s \in S$.

5.4 The zero part $z \in S$ and the one part $y \in S$.

State Transition Functions.

5.5 $\{combine^i\}_{i=0}^{r}$ is a family of functions which model the behavior of a node where r conductance paths join. The order of magnitude approximation, where the strongest signal always dominates all weaker signals, is used. In the following, we will drop the superscript i when we refer to the function.

$$combine \in S^{r+1} \rightarrow S$$
$$combine \equiv \lambda(s_0, s_1 \ldots, s_r) \equiv \max(s_0, s_1, \ldots, s_r)$$

5.6 The functions *initial*0 and *initial*1 set up the signal strength according to the voltage on the node for computing the zero part and the one part, respectively.

$$initial0 \in [S \times \mathcal{V} \rightarrow S]$$
$$initial0(s, v) \equiv \begin{cases} v = \hat{1} \rightarrow 0 \\ (v = \hat{0}) \vee (v = \mathrm{X}) \rightarrow s \end{cases}$$
$$initial1 \in [S \times \mathcal{V} \rightarrow S]$$
$$initial1(s, v) \equiv \begin{cases} v = \hat{0} \rightarrow 0 \\ (v = \hat{1} \vee v = \mathrm{X}) \rightarrow s \end{cases}$$

5.7 A family of functions $\{combinevalue\}_{i=0}^{r}$ models the approximation that the node voltage is determined by the path with the greatest admittance (the dominating path). Signals are propagated step by step through each circuit component in the switch level model. A

signal not on the dominating path might become the signal on the node at some intermediate step before the steady state of the circuit is reached. If the circuit is acyclic, this faulty signal will eventually be overridden by the dominating one. In general, circuits may be cyclic, and such intermediate signals cannot be overridden. For that reason any signal strength which results from the parallel combination of paths will be ignored (set to 0) if it is weaker than the equivalent admittance a.

$$combinevalue \in [S^{r+2} \rightarrow S]$$

$$combinevalue(a, s_0, s_1 \ldots, s_r) \equiv \begin{cases} combine(s_0, s_1 \ldots, s_r) < a \rightarrow 0 \\ combine(s_0, s_1 \ldots, s_r) \geq a \rightarrow \\ \qquad combine(s_0, s_1 \ldots, s_r) \end{cases}$$

5.8 The function *computevalue* combines the zero part and the one part. The equivalent admittance a is obtained assuming the minimum conductance. For transistors with X on their gate, the zero part and one part are obtained assuming maximum conductance. Since the equivalent admittance a must have originated on either a $\hat{1}$ or a $\hat{0}$, at least one of y and z must be at least as large as a. If one of y and z is smaller than a, we can be sure that the larger one results from a dominant admittance path even if all of the transistors with X on their gate are treated as off. Hence the node is given the value corresponding to the larger one. On the other hand if both y and z are as large as or larger than a, we cannot be sure that there will not be an equal strength path to both 1 and 0 for some choice of on or off for transistors with X on their gates. Hence under these conditions the node is given the value X. This mechanism also assures that the

connection of two nodes with equal capacitance strength and different stored values will generate an X.

$$computevalue \in [S^2 \to \mathcal{V}]$$
$$computevalue(s_0, s_1) \equiv \begin{cases} s_0 = 0 \wedge s_1 > 0 \to \hat{0} \\ s_0 > 0 \wedge s_1 = 0 \to \hat{1} \\ else \to \mathbf{X} \end{cases}$$

5.9 The function *update* models the dynamic storage of charge on the capacitance. A signal can be stored on a capacitance c but its strength s can only be as large as c.

$$update \in [S \times C \to S]$$
$$update(s, c) \equiv \min(s, c)$$

6. Conductance Network as a Process

States. A conductance network is composed of conductances (approximation of the transistors) and nodes. It is therefore a *composition* of processes and not a *primitive* process. The states of this composite process are the collection of states of each individual component. We describe each collection by a function from the space-time domain defined below.

$\mathcal{N}_1 \equiv \{1, 2, \ldots, n\}$ is the set which indexes over the nodes in a network.

Let $\mathcal{I}$ and $\mathcal{X}$ be two disjoint subsets of $\mathcal{N}_1$. Set $\mathcal{I}$ indexes over the input and source nodes (these are nodes that provide signals). Set $\mathcal{X}$ indexes over internal nodes (those which provide no signal but are capable of storing charge dynamically). Let $\mathcal{N}_{tran}$ be the set of transistors of the network. Each transistor can be uniquely identified with the triple (i, j, k) where i, j, and $k \in \mathcal{N}_1$ denote nodes at the source, drain and gate of the transistor. Hence a

connectivity function

$$f \in [\mathcal{N}_{tran} \rightarrow \mathcal{N}_1{}^3]$$

which maps the transistors into their unique identification can be defined. Then

$$\mathcal{N}_2 \equiv f(\mathcal{N}_{tran}) \subset \mathcal{N}_1{}^3$$

is the set that indexes over transistors. The space time domain is $\mathcal{N}_1 \times \mathcal{T}$, for nodes and $\mathcal{N}_2 \times \mathcal{T}$ for transistors. The states are described by the following functions from the space-time domain.

6.1 Unknown streams **a**, **z** and **y** in $[\mathcal{N}_1 \times \mathcal{T} \rightarrow \mathcal{S}]$ are the collection of the equivalent admittance, zero part and one part of all nodes, respectively.

6.2 Streams **sr** and **dr** are elements of $[\mathcal{N}_2 \times \mathcal{T} \rightarrow \mathcal{S}]$ which are collections of signal strengths at the source and the drain of the transistors.

6.3 G_0, G_1, and G_2 which are constant functions with respect to time t and therefore are elements of $[\mathcal{N}_2 \rightarrow \mathcal{G}]$.

6.4 C is an element of $[\mathcal{N}_1 \rightarrow \mathcal{C}]$ which is the collection of the capacitance strengths of all nodes.

6.5 V$\in [\mathcal{X} \rightarrow \mathcal{V}]$, and S$\in [\mathcal{X} \rightarrow \mathcal{S}]$. which are the collections of initial voltage and strength of the signals on all internal nodes and are constant functions with respect to t at this level. The strength of input nodes are defined by C, the capacitance strength above. For sources, this strength is c_κ, the greatest element of signal strength $\mathcal{S}$. Variables $in_1, in_2, \ldots, in_p$ are for input nodes of the network and their values will be assigned by the environment.

6.6 A, Z, and Y $\in [\mathcal{N}_1 \rightarrow S]$ are the collection of admittances, zero parts and one parts of the network at steady state defined below.

Algorithms defining State Transition Functions. The state transition functions for the primitive processes (transistors and nodes) define the outputs and next state directly in terms of inputs and current states. For a composite process, outputs and next state usually cannot be defined directly; rather they are defined recursively. The state transition function which relates outputs and next state to inputs and current state is the fixed point of the recursion equations describing the network. The following are the recursion equations for computing the equivalent admittance at each node for an arbitrary conductance network.

6.7 Compute the equivalent admittance at each node.

$$\mathbf{a}(x,t) = \begin{cases} x \in I \rightarrow \mathrm{C}(x) \\ x \in \mathcal{X} \rightarrow \begin{cases} t = 0 \rightarrow \mathrm{S}(x) \\ t > 0 \wedge \mathrm{even}(t) \rightarrow \\ \qquad \mathit{combine}(\mathbf{a}(x,t-1), \mathbf{sr}((x,j_1,k_1),t-1),\ldots, \\ \qquad\quad \mathbf{sr}((x,j_l,k_l),t-1), \mathbf{dr}((i_1,x,k_{l+1},t-1),\ldots, \\ \qquad\quad \mathbf{dr}(i_r,x,k_{l+r},t-1)) \\ \mathrm{odd}(t) \rightarrow \mathbf{a}(x,t-1) \end{cases} \end{cases} \tag{1}$$

where l is the number of transistors whose source is node x and r is the number of transistors whose drain is node x. For the sake of clarity, we use $j_1,\ldots,j_l$, $i_1,\ldots,i_r$ and $k-1,\ldots,k_{l+r}$ to express the neighboring nodes of node x (since these nodes are fixed for a given network), instead of using the connectivity function f mentioned above to express these nodes as an explicit function of x.

$$\mathbf{sr}((i,j,k),t) = \begin{cases} \text{odd}(t) \to \mathit{couple}(\mathbf{a}((i,j,k) \circ 2, t-1), \mathrm{G}_1(i,j,k)) \\ \text{even}(t) \wedge t > 0 \to \mathbf{sr}((i,j,k), t-1) \\ t = 0 \to \perp \end{cases} \tag{2}$$

$$\mathbf{dr}((i,j,k),t) = \begin{cases} \text{odd}(t) \to \mathit{couple}(\mathbf{a}((i,j,k) \circ 1, t-1), \mathrm{G}_1(i,j,k)) \\ \text{even}(t) \wedge t > 0 \to \mathbf{dr}((i,j,k), t-1) \\ t = 0 \to \perp \end{cases} \tag{3}$$

Substituting (2) and (3) into (1) we have

$$\mathbf{a}(x,t) = \begin{cases} x \in I \to \mathrm{C}(x) \\ x \in \mathcal{X} \to \begin{cases} t = 0 \to \mathrm{S}(x) \\ t > 0 \wedge \text{even}(t) \to \mathit{combine}(\mathbf{a}(x, t-1), \\ \qquad \mathit{couple}(\mathbf{a}(j_1, t-2), \mathrm{G}_1(x, j_1, k_1)), \ldots, \\ \qquad \mathit{couple}(\mathbf{a}(j_l, t-2), \mathrm{G}_1(x, j_l, k_l)), \\ \qquad \mathit{couple}(\mathbf{a}(i_1, t-2), \mathrm{G}_1(i_1, x, k_{l+1})), \ldots, \\ \qquad \mathit{couple}(\mathbf{a}(i_r, t-2), \mathrm{G}_1(i_r, x, k_{l+r}))) \\ \text{odd}(t) \to \mathbf{a}(x, t-1) \end{cases} \end{cases} \tag{4}$$

Notice that when t is odd, $\mathbf{a}$ is exactly the same as it was at $t-1$, and therefore it is redundant to specify $\mathbf{a}$ at odd t. Hence (4) can be written as

$$\mathbf{a}(x,t) = \begin{cases} x \in I \to \mathrm{C}(x) \\ x \in \mathcal{X} \to \begin{cases} t = 0 \to \mathrm{S}(x) \\ t > 0 \to \mathit{combine}(\mathbf{a}(x, t-1), \\ \qquad \mathit{couple}(\mathbf{a}(j_1, t-1), \mathrm{G}_1(x, j_1, k_1)) \ldots, \\ \qquad \mathit{couple}(\mathbf{a}(j_l, t-1), \mathrm{G}_1(x, j_l, k_l)), \\ \qquad \mathit{couple}(\mathbf{a}(i_1, t-1), \mathrm{G}_1(i_1, x, k_{l+1})), \ldots, \\ \qquad \mathit{couple}(\mathbf{a}(i_r, t-1), \mathrm{G}_1(i_r, x, k_{i+r}))) \end{cases} \end{cases} \tag{5}$$

This equation defines the equivalent admittance for each node in terms of the equivalent admittance of all nodes adjacent to it. For an isolated node,

the equivalent admittance is just the node capacitance strength. For a group of nodes which are not connected by any conductance paths to sources, the equivalent capacitances for these nodes are computed. The phenomenon of charge sharing occurs in this situation and is handled properly by the same mechanism which handles the combination of paths. Function **a** specifies the equivalent admittance for any node at any time. Since the functional in (5) is continuous, **a** is well defined by the fixed-point theorem. Notice that the time domain is infinite, thus obtaining **a** takes an infinite number of time steps. Fortunately, most circuits reach a ***steady state*** in a finite subset of the time domain. In our framework, a process reaches its steady state if all of its states do not change after a certain time has elapsed. Using a specific process as an example, the collection of equivalent admittances of a conductance network becomes steady at time ξ_a if for every node in $\mathcal{N}_1$, the admittance a is the same as it was at one time step earlier, i.e.,

$\mu t[\lambda x.(a(x,t)-a(x,t-1)) = \lambda x.0] = \xi_a$, where μ is the minimalization operator.

This operator μ gives the smallest t such that $\lambda x.(a(x,t) - a(x,t-1)) = \lambda x.0$. We can now restrict the time domain to $\mathcal{T}_{\xi_a} \equiv \{0,1,2\ldots,\xi_a\}$. The equivalent admittance on all nodes can then be described by

$\mathbf{A} \in [\mathcal{N}_1 \rightarrow \mathcal{S}]$, $\mathbf{A}(x) \equiv \mathbf{a}^{\xi_a}(x,\xi_a)$ where a^{ξ_a} is the least fixed point of (5).

6.8 Compute the zero part of the network. The equations are similar to the ones above except that the maximum conductance G_2 is used instead of G_1, and the equivalent admittance A obtained above is used.

$$\mathbf{z}(x,t) = \begin{cases} x \in I \to \mathit{initial0}(\mathrm{C}(x), in_x) \\ x \in \mathcal{X} \to \begin{cases} t = 0 \to \mathit{initial0}(\mathrm{S}(x), \mathrm{V}(x)) \\ t > 0 \wedge \mathrm{even}(t) \to \mathit{combinevalue}(\mathrm{A}(x), \mathbf{z}(x, t-1), \\ \qquad \mathbf{sr}((x, j_1, k_1), t-1), \ldots, \mathbf{sr}((x, j_l, k_l), t-1), \\ \qquad \mathbf{dr}((i_1, x, k_{l+1}, t-1), \ldots, \mathbf{dr}(i_r, x, k_{l+r}, t-1))) \\ \mathrm{odd}(t) \to \mathbf{z}(x, t-1) \end{cases} \end{cases} \tag{6}$$

$$\mathbf{sr}((i,j,k),t) = \begin{cases} \mathrm{odd}(t) \to \mathit{couple}(\mathbf{z}((i,j,k) \circ 2, t-1), \mathrm{G}_2(i,j,k)) \\ \mathrm{even}(t) \wedge t > 0 \to \mathbf{sr}((i,j,k), t-1) \\ t = 0 \to \perp \end{cases} \tag{7}$$

$$\mathbf{dr}((i,j,k),t) = \begin{cases} \mathrm{odd}(t) \to \mathit{couple}(\mathbf{z}((i,j,k) \circ 1, t-1), \mathrm{G}_2(i,j,k)) \\ \mathrm{even}(t) \wedge t > 0 \to \mathbf{dr}((i,j,k), t-1) \\ t = 0 \to \perp \end{cases} \tag{8}$$

Following the same steps as above and substituting (7) and (8) into (6), we have

$$\mathbf{z}(x,t) = \begin{cases} x \in I \to \mathit{initial0}(\mathrm{C}(x), in_x) \\ x \in \mathcal{X} \to \begin{cases} t = 0 \to \mathit{initial0}(\mathrm{S}(x), \mathrm{V}(x)) \\ t > 0 \to \mathit{combinevalue}(\mathrm{A}(x), (\mathbf{z}(x, t-1), \\ \qquad \mathit{couple}(\mathbf{z}(j_1, t-1), \mathrm{G}_2(x, j_1, k_1)) \ldots, \\ \qquad \mathit{couple}(\mathbf{z}(j_l, t-1), \mathrm{G}_2(x, j_l, k_l)), \\ \qquad \mathit{couple}(\mathbf{z}(i_1, t-1), \mathrm{G}_2(i_1, x, k_{l+1})), \ldots, \\ \qquad \mathit{couple}(\mathbf{z}(i_r, t-1), \mathrm{G}_2(i_r, x, k_{l+r}))) \end{cases} \end{cases} \tag{9}$$

Similarly, the least fixed-point of the equation is an element of $[\mathcal{N}_1 \times \mathcal{T}_{\xi_x} \to S]$, where $\mathcal{T}_{\xi_x}$ is the restricted time domain and the steady state zero part strength on all nodes is described by

$$\mathrm{Z} \in [\mathcal{N}_1 \to S], \quad \mathrm{Z}(x) \equiv \mathbf{z}^{\xi_x}(x, \xi_x).$$

6.9 Compute the one part of the network. Similar to equation (9), we have

$$\mathbf{y}(x,t) = \begin{cases} x \in I \rightarrow \mathit{initial1}(\mathrm{C}(x), in_x) \\ x \in \mathcal{X} \rightarrow \begin{cases} t = 0 \rightarrow \mathit{initial1}(\mathrm{S}(x), \mathrm{V}(x)) \\ t > 0 \rightarrow \mathit{combinevalue}(\mathrm{A}(x), (\mathbf{a}(x, t-1), \\ \qquad \mathit{couple}(\mathbf{y}(j_1, t-1), \mathrm{G}_2(x, j_1, k_1)) \ldots, \\ \qquad \mathit{couple}(\mathbf{y}(j_l, t-1), \mathrm{G}_2(x, j_l, k_l)), \\ \qquad \mathit{couple}(\mathbf{y}(i_1, t-1), \mathrm{G}_2(i_1, x, k_{l+1})), \ldots, \\ \qquad \mathit{couple}(\mathbf{y}(i_r, t-1), \mathrm{G}_2(i_r, x, k_{l+r}))) \end{cases} \end{cases} \tag{10}$$

The one part of the network can be obtained as before, and is described by

$$\mathrm{Y} \in [\mathcal{N}_1 \rightarrow \mathcal{S}], \quad \mathrm{Y}(x) \equiv \mathbf{y}^{\xi_y}(x, \xi_y)$$

where $\xi_y = \mu t[\lambda x.(y(x,t) - y(x,t-1))x.0]$, and y^{ξ_y} is the least fixed point of(10).

State Transition Functions. The state transition functions are least fixed-points of above algorithms.

$$\begin{aligned} &\mathit{Eqv} \in [[\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow \mathcal{S}] \times [\mathcal{N}_2 \rightarrow \mathcal{G}] \rightarrow [\mathcal{N}_1 \rightarrow \mathcal{S}]] \\ &\mathit{Eqv}(\mathrm{V}, \mathrm{S}, \mathrm{G}) \equiv \mathrm{A} \\ &\mathit{Zero}, \mathit{One} \in [[\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow \mathcal{S}]^2 \times [\mathcal{N}_2 \rightarrow \mathcal{G}] \rightarrow [\mathcal{N}_1 \rightarrow \mathcal{S}]] \\ &\mathit{Zero}(\mathrm{V}, \mathrm{S}, \mathrm{A}, \mathrm{G}) \equiv \mathrm{Z} \\ &\mathit{One}(\mathrm{V}, \mathrm{S}, \mathrm{A}, \mathrm{G}) \equiv \mathrm{Y} \end{aligned}$$

6.10 With the above functions, the node voltage for the conductance network can be obtained by the following function:

$$\begin{aligned} &\mathit{Cnetwork} \in [[\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow \mathcal{S}] \times [\mathcal{N}_2 \rightarrow \mathcal{G}]^2 \rightarrow [\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow \mathcal{S}]] \\ &\mathit{Cnetwork}(\mathrm{V}, \mathrm{S}, \mathrm{G}_1, \mathrm{G}_2) \equiv (\lambda x.\mathit{computevalue}(\mathit{Zero}(\mathrm{V}, \mathrm{S}, \mathit{Eqv}(\mathrm{V}, \mathrm{S}, \mathrm{G}_1), \mathrm{G}_2)(x), \\ &\qquad \mathit{One}(\mathrm{V}, \mathrm{S}, \mathit{Eqv}(\mathrm{V}, \mathrm{S}, \mathrm{G}_1), \mathrm{G}_2)(x))), \quad \mathit{Eqv}(\mathrm{V}, \mathrm{S}, \mathrm{G}_1)) \end{aligned}$$

Formation of Conductance Network

6.11 Compute the connectivity of the network (network topology) which is given by those transistors that are on. There are two possible extremal network topologies; one is given by assuming maximum conductance and the other by assuming minimum conductance. The state transition function for computing the topology of each conductance network is defined as

$$Topo \in [[\mathcal{N}_1 \to \mathcal{V}] \times [\mathcal{N}_2 \to P] \times [\mathcal{N}_2 \to \mathcal{G}] \to [\mathcal{N}_2 \to \mathcal{G}]^2]$$
$$Topo(\mathrm{V}, \mathrm{P}, \mathrm{G}_0) \equiv (\lambda(i,j,k).setconduc_1(\mathrm{V}(k), \mathrm{P}(i,j,k), \mathrm{G}_0(i,j,k)),$$
$$\lambda(i,j,k).setconduc_2(\mathrm{V}(k), \mathrm{P}(i,j,k), \mathrm{G}_0(i,j,k))$$
$$)$$

6.12 *Dstore* models the dynamically stored charge on each node by setting the strength of the signal stored on each node to the capacitance strength of that node.

$$Dstore \in [[\mathcal{N}_1 \to S] \times [\mathcal{N}_1 \to C] \to [\mathcal{N}_1 \to S]]$$
$$Dstore(\mathrm{S}, \mathrm{C}) \equiv \lambda x.update(\mathrm{S}(x), \mathrm{C}(x))$$

7. Transistor Network as a Process

After obtaining the functions for a conductance network and the formation of a new conductance network, a space-time algorithm which describes a transistor network can be given.

Algorithm for a Transistor Network. The following equation describes how the topology of each of the successive conductance network is constructed from the signals computed by the previous conductance network. Each of these conductance networks computes its signal values from the dynamically stored signals of the previous network. Notice that each time step at this level corresponds to $\xi_a + \xi_x + \xi_y$ finer steps at the conductance

network level.

$$(V,S)(t) = \begin{cases} t = 0 \rightarrow (V_0, S_0) \\ t > 0 \rightarrow Cnetwork(V(t-1), Dstore(S(t-1), C_0), \\ \qquad\qquad Topo(V(t-1), P_0, G_0)) \end{cases} \tag{11}$$

The least fixed-point of the functional in (11) is denoted by

$$(V,S)^{\xi} \in [\mathcal{T}_{\xi} \rightarrow [\mathcal{N}_1 \rightarrow S] \times [\mathcal{N}_1 \rightarrow \mathcal{V}]]$$
where $\xi = \mu t[V(t) - V(t-1) = \lambda x.0 \wedge S(t) - S(t-1) = \lambda x.0]$
and $\mathcal{T}_{\xi}$ is the restricted time domain at this level.

We further λ-abstract the appropriate arguments to obtain the following function which models the behavior of a transistor network,

$$Tnetwork \in [[\mathcal{N}_1 \rightarrow P] \times [\mathcal{N}_1 \rightarrow \mathcal{G}] \times [\mathcal{N}_2 \rightarrow C] \times [\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow S] \rightarrow [\mathcal{T}_{\xi} \rightarrow [\mathcal{N}_1 \rightarrow \mathcal{V}] \times [\mathcal{N}_1 \rightarrow S]]]$$
$$Tnetwork(P_0, G_0, C_0, V_0, S_0) \equiv (V,S)^{\xi}.$$

For a given set of transistors (P_0 specifies the types and G_0 the conductance strength) and nodes (C_0 specifies the node capacitance strength), the connection among the nodes (defines $\mathcal{N}_2$), and the initial state of the network (V_0 and S_0), the function gives the behavior of the network in time (each step being a conductance network). The steady state signal values for all nodes are given by

$$(\mathrm{V}, \mathrm{S}) \in [\mathcal{N}_1 \rightarrow \mathcal{V} \times S]$$
$$(\mathrm{V}, \mathrm{S}) \equiv Tnetwork(P_0, G_0, C_0, S_0, V_0)(\xi).$$

Let $in_1, in_2, \ldots, in_p \in I$ be the input nodes and $o_1, o_2, \ldots, o_q \in \mathcal{X}$ be the output nodes, then the functional description of the network is

$$net \in [[\mathcal{V} \times S]^p \rightarrow [\mathcal{V} \times S]^q]$$
$$net \equiv (net_1, net_2, \ldots, net_q)$$
where $net_i(in_1, \ldots, in_p) \equiv Tnetwork(P_0, G_0, C_0, S_0, V_0)(\xi)(o_i)$
and $i = 1, 2, \ldots, q$.

Let the new time step t' be ξ time steps at the transistor network level. At this point, the behavior of the network is obtained. It is a state transition function from inputs $(in_1, \ldots, in_p)$ and initial state at $t = 0$ (also $t' = 0$) to outputs $(o_1, \ldots, o_q)$ and next state (the steady state at the transistor network level) at $t' = 1$ $(t = \xi)$.

8. Functional Abstraction and Semantic Hierarchy

In describing a transistor network, three hierarchical levels are encountered, namely, the transistor and node level, conductance network level and the transistor network level. In order to approach the design hierarchically or raise the design to the level of functional blocks, the entire transistor network must be partitioned into pieces where the behavior of each piece can be obtained and thereafter abstracted as a state transition function. The partition is not arbitrary; it only makes sense to partition in such a way that the behavior of the algorithm with the partition (with or without an equivalent functional abstraction) is equivalent to that of the original algorithm without the partion (and abstraction). When a partition satisfies this criterion, a new hierarchical level is created. We call it a *semantic level.* There are ways to partition a design that do not satisfy the criterion but are useful in the physical layout of a design. Any level created by these partitions is termed a *syntactic level.*

9. Data Abstraction

Aside from the functional abstraction achieved by taking the least fixed-point of an algorithm, data abstraction often occurs. By the definition of *Cnetwork*, we can see that if all inputs to a transistor network are the gates, the strengths of these input signals are immaterial since only the strengths of dynamic stored charges are used. Therefore the strength part of signal can be ignored at *gate* level and the data type become $\mathcal{V}$ instead of $\mathcal{V} \times \mathcal{S}$.

In the next chapter, different semantic hierarchies are defined so that data abstraction can be performed at each level. If a transistor network is clocked (by either a clock or self-timed signaling), the data type can be further abstracted to be B the boolean domain rather than the Ternary logic values. Once the *clocked cell* level of abstraction is achieved, the manipulation of functional definitions of circuits is in the familiar regime where boolean algebra or the usual algebra on integers can be applied. Formal verification can be done using various inductive techniques [43] such as structural induction [5]. Since a clocked cell is usually fairly small, an exhaustive simulation of the cell is more convenient than formal manipulation of its description by hand. A hierarchical simulator based on the space-time algorithms and their semantics will be described in the next chapter.

Chapter 5

A Hierarchical Simulator

Simulation consists of exercising the representation of a design on a general purpose computer. It differs from programming only because the ultimate implementation will be in a different medium, say a VLSI chip. In order for simulation to be in any sense effective, the simulated system must perform the same function as the ultimate implementation. A VLSI chip is a highly concurrent object; the simulation of such a chip amounts to programming a highly concurrent system. It follows that any demonstrably correct simulation technique will be one of the two types:

(i) The entire design is represented as an implementation with objects which are abstract models of the medium at the bottom level (e.g. transistor model). The simulation operates on a representation which is a direct image of the fully instantiated implementation in the medium.

(ii) The design is represented as a hierarchy of implementations. Each level of implementation is constructed of objects which are abstract models of the implementation at the level below it. The simulation operates on a hierarchical representation where each level is refined by the level below it.

The first approach requires only a model of the implementation medium. The second approach requires, in addition, a general principle for obtaining

an abstract model from a given implementation of objects at the lower level. The second approach allows the implementation details to be hidden and therefore yields a clearer conceptualization of the design and a more efficient simulation.

1. Multi-level and Mixed-level Simulation

In the design of a VLSI system, the traditional levels of hierarchy are circuit level, gate-level, and register transfer level. This partitioning helps designers focus on one particular level of design at any given time. When they focus on the register transfer level, for example, they can reason about the overall design in terms of the functionality of the inter-connected blocks and a given timing scheme, without worrying about the details inside each block. On the other hand, when they are designing at the circuit-level, the focus is on one functional block at a time rather than on the whole system. Hierarchical simulators such as VISTA [12] and [31] allow designers to focus on one part of the design in this way. Ideally, if the overall system design is shown to be correct in terms of the functional blocks, and each functional block is shown to be correct in terms of its circuit-level or gate-level implementation, the designers need not examine the correctness of the detailed implementation across two different functional blocks; i.e., each of these hierarchical levels provides an abstraction of the level below it. The functionality of the overall design will always be preserved when the designers cross the different levels. The complexity of a large system design can only be effectively managed through these levels of abstraction. Preserving the functionality, i.e., maintaining consistency between hierarchical levels, is the most important property of a hierarchical simulator, and the most difficult to achieve. Successful treatment of the consistency problem has not been found in the literature.

A multi-level simulator, when used as a tool to verify a hierarchical design, should provide a way to ensure the consistency of the design throughout all levels. On the other hand, the simulator should allow blocks of different levels to be connected through proper interfaces which handle the timing and the matching of various input/output data types. The key issue in such a multi-level simulator is the interface mechanism. In this chapter, we present a simulator in which a uniform representation is used at all levels of the design. A method of abstraction for maintaining consistency between levels and proper interfacing of timing and data types is described.

2. Semantic Hierarchy and Syntactic Hierarchy

In conventional programming languages, *macros* and *procedures* or *functions* have long been recognized as two different ways to facilitate programming. Macros are used only at a syntactic level to ease the specification, and do not provide any semantic abstraction, since they are expanded during compilation. The object code of a program using macros is exactly the same as its counterpart without using macros. However, procedures and functions are used not only to facilitate specification, but to encapsulate a piece of code with a well-defined interface to other parts of a program. Ideally, a function should not allow any side-effects and therefore provides a semantic abstraction. We make the distinction of the *syntactic hierarchy* vs. the *semantic hierarchy* in a simulator in a way analogous to the distinction between macros and procedures in a programming language. The syntactic hierarchy in the simulator serves two purposes: One is ease of specification, just as macros in a programming language; the other is that it contains information about spatial locality. Since it has been observed that activities in circuits tend to be local [3], this information can be exploited by the simulation algorithm to achieve better performance. Unlike simulators which take a flat

network specification where the locality information has been thrown away by the preprocessing to be recovered later on by topological analysis, this simulator takes advantage of user's hierarchical specification and requires neither preprocessing nor topological analysis.

Syntactic Cells. In the context of a switch-level simulator, transistors and nodes are objects from which everything else is constructed. They are bottom level cells of the semantic hierarchy. We call this bottom level the *conductance-level* since an active device (transistor) is approximated by a passive conductance. A *syntactic cell* is a circuit consisting of an interconnection of transistors and nodes where there is no restriction on the input nodes or the output nodes of the cell. The state of a node in a syntactic cell can directly or indirectly influence nodes in the others and thus the cell provides no abstraction for the behavior of a circuit.

Figure 5–1 shows an nMOS *exclusive nor* cell which is an example of a syntactic cell.

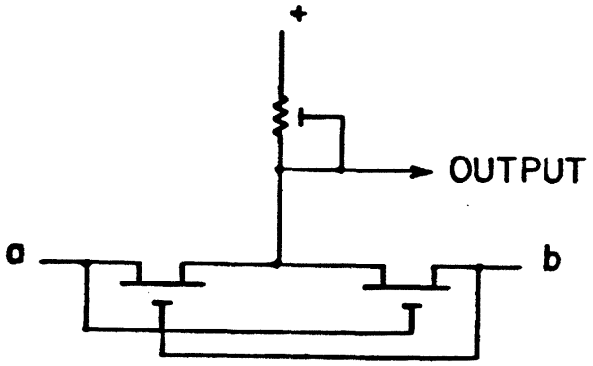


Figure 5–1 A non-restoring exclusive nor circuit XNOR.

Although a syntactic cell like the one shown in Figure 5–1 is a composition of transistors and nodes, it is at the same semantic level (or of the

same semantic type) as transistors and nodes. A syntactic cell can also be a composition of other syntactic cells of the same semantic type, although these cells can be nested at an arbitrary depth in the syntactic hierarchy. No separation of the hierarchy into leaf cells and composition cells [35] is required, i.e., a transistor or a node can be composed with a syntactic cell directly without making it into a syntactic cell by itself. The simulation of such a cell produces exactly the same result as the circuit represented without hierarchy. The equivalence of Bryant's model of a flat network with our hierarchical represented network can be shown by straightforward induction on the level of the syntactic hierarchy.

The semantic hierarchy is constructed for abstracting the behavior of circuits. A syntactic cell is made into a semantic cell if an abstraction of its behavior is desired. This new *primitive* semantic cell is used as an "atom" in a new syntactic hierarchy, which in turn is used within each semantic level in order to clarify the specification and express the locality of cells.

For simulator based on a switch-level model, the semantic levels can be the bottom level transistors and nodes (conductance-level), gate-level, clocked cell level (in a more general sense including a self-timed [39l] module with request and acknowledge signals), register transfer level, and other higher levels. We call all levels at and above the clocked cell level the ***functional level*** since the functionality of cells at those levels can be abstracted.

Gate-level Cells. A *gate-level* cell is a composition of conductance-level cells (transistors, nodes and syntactic cells composed of them) with the restriction that the input nodes be uni-directional, i.e., the input nodes are gates of transistors. Figure 5–2 shows a gate-level cell. Note that the XNOR circuit in Figure 5–1 is not a gate-level cell.

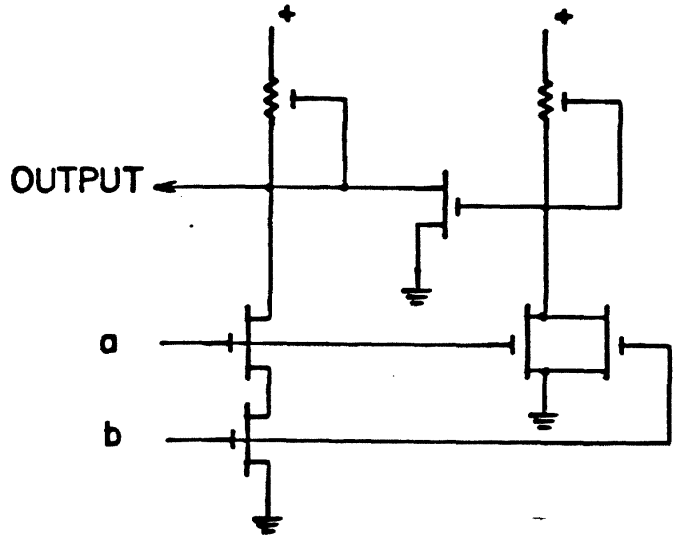


Figure 5–2 A restoring exclusive or gate XOR.

According to the unit-delay model of a transistor, the gate node of a transistor will not affect the state of the transistor until a given conductance network is settled. Since each of the input nodes of a gate-level cell is the gate node of a transistor, no intermediate state on that node will be seen by the cell until the node reaches the inner conductance-level steady state described above. The gate-level cell is affected by each of its inputs when each of these gate nodes reaches its steady state and causes the corresponding transistor to change state. Notice that to abstract the behavior of a gate-level cell, we can only discard the intermediate states on all of its output nodes (prior to its reaching the inner conductance-level steady state). We cannot just keep the outer transistor-level steady state and discard all the conductance level steady states. Therefore, the conventional way of thinking about a gate-level cell as a functional block (in which all intermediate states before reaching the outer transistor-level steady-state are discarded) is not formally correct. This incorrect abstraction must be remedied by some other analysis in the design process. For a simulator, such incorrect abstraction will miss, for example,

the glitch in the circuit shown in Figure 5–3.

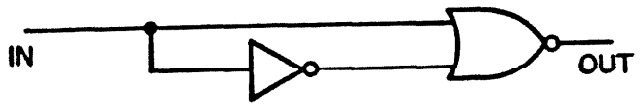


Figure 5–3 A gate-level circuit that generates logic 0

If we reason about the circuit shown in Figure 5–3 at the functional level, the output of this cell will be a logic 0 for all possible inputs. Yet when the input is initially 1 and switches to 0, the output will become a logic 1 before it settles back to 0. If the inner conductance-level steady state is kept, the simulation output will reflect the glitch properly. In actual design practice, one often reasons about interconnections of gate-level cells which behave like the above example, even though the functional abstraction is not formally correct. This formally incorrect but practically valuable abstraction works because one adopts a *timing discipline* in composing gate-level cells. The timing discipline ensures that each combinational circuit in a given network can only be affected by the steady state value of the output of the combinational cell to which it is connected. Familiar examples of this discipline are a two phase non-overlapping clock scheme where the clock period of each phase is long enough for a combinational circuit to settle, and a self-timed request-acknowledge signaling. An intermediate value on an output node will not propagate because of the timing discipline in the same way that the intermediate value of a local variable will not be returned by a function in a programming language. Therefore the timing discipline provides a semantic abstraction similar to a function in a programming language.

Clocked Cells. A clocked-cell is a composition of conductance level cells, with the restriction that all input nodes must satisfy a timing discipline

which insures that only the steady-state of the output nodes can be seen by other cells to which they are connected. (Figure 5–4 shows a clocked cell formed by two other clocked cells.)

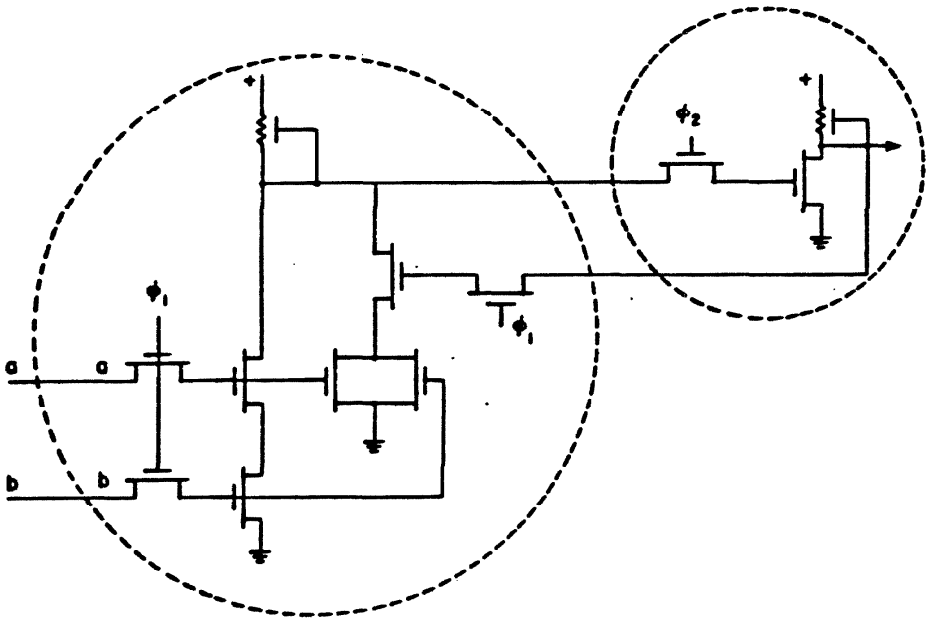


Figure 5–4 A clocked Muller C-element containing two sub-cells

The steady state of a primitive clocked cell (which does not contain any other clocked cell as a component, for example, as each of the sub-cells shown in Figure 5–4) is the steady state at transistor-level for that cell. Each of these cells can only see the steady state of other cells and therefore provides an abstraction. The gate-level abstraction is at a lower level than the clocked cell level since not only the steady state of the transistor-level is kept but all the steady states at the conductance level are kept as well. Bryant [3] points out that the gate-level cells provide a useful modeling abstraction since it is not necessary to keep track of the ***signal strength*** used in the conductance-level.

Experience with MOS design has shown that specifying a chip in terms of gate-levels cells is not convenient because the restriction on inputs does not allow effective use of pass transistors. Since clocked cells are usually small, we use them as the semantic level immediately above the conductance-level cells. Sub-circuits within them are represented as syntactic cells. In this simulator we therefore do not support the gate-level abstraction. In a technology other than MOS, the gate-level may well be an appropriate level of abstraction, and is very easy to implement within our simulator.

We will now proceed to illustrate the model and its use in specification and simulation by way of a simple example — a pipelined inner product element similar to that described in [25]. Figure 5–5 shows an example of a hierarchical partition of a single bit inner product cell IPB described in [44] which will be used later in the pipelined inner product element.

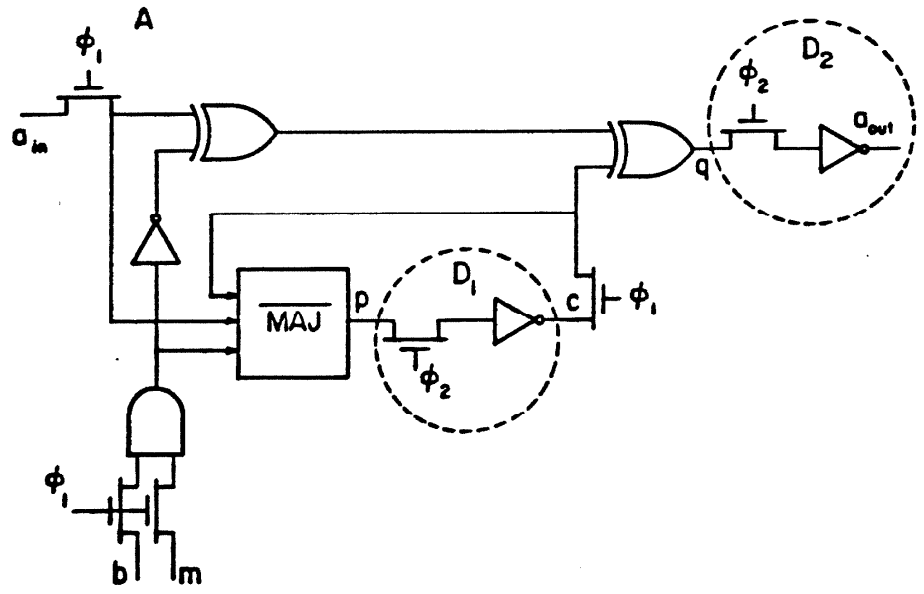


Figure 5–5 A single bit inner product element IPB

This cell is of type clocked-cell and consist of three clocked-cells, A which is not circled in the Figure, D_1 and D_2 which are circled. Cell A contains five syntactic cells, $\overline{MAJ}$, two XNOR gates, an AND gate, and an inverter. It also contains four transistors and four gate nodes each with ϕ_1 on it, four input nodes a_{in}, b, m and, c and finally, two output nodes p and q.

Cells D_1 and D_2 are identical. Each consists of one syntactic cell, the inverter, a transistor and a gate node with ϕ_2 on it, an input node p for D_1 and q for D_2, respectively and finally an output node c for D_1 and a_{out} for D_2, respectively. The inputs to the IPB are a_{in}, b, m and the clocks. The output is a_{out}. There is one bit of internal state in this cell, namely the carry c. The current state of c is denoted by c_{cur} and next state by c_{next}.

The simulation of this syntactic cell proceeds by (1) obtaining the steady state of each clocked-cell (A, D_1 and D_2) using its inputs and state, independently of the other cells and, (2) transferring the outputs of one cell to the inputs of the others. Notice that a IPB cell is a syntactic cell, and therefore each of the three sub-cells is only invoked once. The IPB cell can be further abstracted to be a bit-level cell. However, it is then necessary to obtain the steady state of the IPB cell. The above procedure is iterated until the outputs of all three sub-cells do not change any more. This iteration yields the steady state of the IPB cell at the bit-level. This example shows that each semantic level requires an iteration to obtain its steady state. In the example of constructing an n bit pipelined element below, we do not use the bit-level abstraction, and so the above described procedure is invoked only once.

To verify the correctness of this circuit, the three clocked-cells are replaced by their functional specification, shown in Figure 5–6. The cell

specified in a functional form is simulated and compared with the one (with detailed implementation) described above.

```
IF φ1 THEN
BEGIN
      q ← (a_in ⊕ (NOT(b AND m))) ⊕ c_cur ,
      p ← MAJ(a_in, b AND m, c_cur) ;
END ELSE
IF φ2 THEN
BEGIN
      a_out ← NOT q,          c_next ← NOT p;
END;
```

Figure 5–6 The functional specification of a 1-bit inner product element.

Notice that the data type in the functional specification is ***boolean*** rather than ***signal*** (which has two components, the signal strength and the signal state) used in the conductance-level representation, and that the algebra of signals [3] is different from ordinary boolean algebra. In many cases, such as this inner product cell, a correct signal on the output of each clocked cell can only be a logic 0 or a logic 1 which falls into the domain of boolean algebra, although the internal nodes or even the intermediate state of the outputs can be in state X. The algebra of signals contains the signal state X, which represents an intermediate voltage between 1 and 0. The steady state output of a proper clocked cell will always be a binary value. If the output of a clocked-cell is an X, then either an error has occurred in the implementation or functional abstraction at the clocked cell level should not be applied. It is possible, in fact, very common, for the output of a clocked cell to be ***unknown***, for example, when the output is a function of some state variable which

has not been initialized. At every level we use the symbol $\perp$ (pronounced *bottom*) to represent an undefined value. It must not be confused with X which represents a voltage between logic 0 and logic 1 at the conductance-level. Since an intended output will never depend on an unknown value $\perp$, a possible way of handling a $\perp$ in the input of a function would be to define the corresponding output also to be $\perp$. An algebra extended this way is called *naturally extended* [26]. (Although there exists other possible extensions, the natural extension is used in this simulator.).

Notice that even at the level of a simple 1-bit cell, we can see the kind of data abstraction that always accompanies functional abstraction. Each level has its own algebra for manipulating data and functions. A formal treatment of the model of computation that allows such abstractions (input/output mapping functions for data abstraction and fixed-point semantics for functional abstraction) is given in the previous Chapters.

Word-level Cells. The functional specification given in Figure 5–6 is used in the next level of composition, in this example the pipelined inner product element. The performance of the simulation using this specification will be drastically improved in comparison with the circuit-level specification. Although exhaustive checking is possible for verifying the consistency between the two specifications for a single cell like IPB, it becomes rapidly impractical as the size of the cell grows. Then it becomes necessary for the notation or language in which the design is described to have formal semantics in order to allow verification of the consistency between two different levels of specification. Space-time algorithms are an example of such a notation. Although the current simulator is implemented as an embedded language in an ordinary programming language, the primitives for specifying the design are a direct mapping of the above formal notation.

An n-bit pipelined inner product element can be composed by connecting n IPB cells serially, the specific scheme is shown in Chapter 3. We call this cell IPE. It can be viewed as a word-level cell where a pulse input *lsb* indicates the start of a word (say, least significant bit of an n bit word). We can adapt the interface of this cell so that the clocks can be hidden inside the cell and the bit serial input and output can be abstracted as words. Figure 5–7 shows the data abstraction which maps input words $\hat{a}_{in}$, $\hat{b}$ and $\hat{m}$ into series of bits and collects output bits to be the word $\hat{a}_{out}$. The bits $\hat{a}_{in}[i]$, $\hat{b}[i]$, and $\hat{m}[i]$ are put in one by one to the IPE at its input ports $ipe_{a_{in}}$, ipe_b, and ipe_m. A suitable clock is also generated for the cell. Once the inputs and the clocks are valid, IPE is called upon to compute its result (written as IPE.*compute* in Figure 5–7). It computes by invoking each of the IPB cells, transfers the results of one IPB to the next and repeats until each of the result returned by all IPB cells becomes steady. The individual IPB cell, when invoked, in turn invokes each of its sub-cells A, D_1 and D_2 only one time as described above.

The functional abstraction of the IPE is shown in Figure 5–8. Again, the consistency of the two different levels can be verified by the combination of formal verification and simulating both specifications.

```
IF lsb THEN
FOR i := 0 TO n − 1 DO (loading phase)
BEGIN
      ipe_{a_in} ← â_in[i], ipe_b ← b̂[i], ipe_m ← m̂[i]; (input mapping)
      φ1 ←high, φ2 ←low; IPE.compute;
      φ1 ←low, φ2 ←high; IPE.compute;
END;
FOR i := 0 TO n − 1 DO (unloading phase)
```

BEGIN

ϕ_1 ←**high**, ϕ_2 ←**low**; IPE.*compute*;

ϕ_1 ←**low**, ϕ_2 ←**high**; IPE.*compute*;

$\hat{a}_{out}[i] \leftarrow ipe_{a_{out}}$; **(output mapping)**

END;

Figure 5–7 Interface of an n bit-serial element to higher-level specification.

$$\text{IF } lsb \text{ THEN } \hat{a}_{out} \leftarrow \hat{a}_{in} + \hat{b} \times \hat{m}$$

Figure 5–8 The functional specification of the inner product element.

The n bit inner product element can be used to construct, for example, a systolic array performing matrix multiplication [22]. At the level of a systolic array, the inner product element is used in the functional form as in Figure 5–8, regardless of its implementation as bit serial or word parallel. We have to be careful however, since the mapping functions that constitute the data abstraction are never unique. The mapping may be from n serial bits to a word or from n parallel bits to a word. The mapping in the former is from time domain to an abstract word and the latter is from space domain to an abstract word. In connecting two inner product elements at this level, one must make sure that the output mapping function of one element is the inverse of the input mapping function of the element to which it is connected. Two IPE elements as shown in Figure 5–7 can be connected since the output mapping of one (from bits $ipe_{a_{out}}[i]$ for $0 \leq i < n$ to an n bit word $\hat{a}_{out}$) is the inverse of the input mapping of the other (from n bit word $\hat{a}_{in}$ to bits $ipe_{a_{in}}[i]$ for $0 \leq i < n$). The lower level has the same interface problem in a different form. It is the timing discipline used in the design that allows abstraction to the clocked cell level. This discipline can be one of several kinds. For example it may be two, three or four phase clock, or a two or

four cycle request-acknowledge protocol. Once again the condition imposed on the output of one cell must match that of the input of the cell to which it connects.

3. "Structured Programming" in VLSI

The systolic array can also be abstracted from its implementation and used as an abstract machine performing matrix multiplication, which is presented in Chapter 3. Notice that the abstraction mechanism is precisely the same at all levels. Considering the bottom-up approach, we summarize the two essential steps in the hierarchical design method:

(1) Use a cell defined by its implementation in the context of cells defined by their functional specifications. In order to adapt the interface between the cell and its context, a new cell containing the following three parts is constructed:

i) A function which maps inputs in the data representation of the high level to the inputs at the lower level.

ii) The implementation at the lower level.

iii) A function which maps the outputs at the lower level to outputs in the data representation at the higher level.

This cell is now a proper cell with exactly the same interface as a higher level cell. We can use it as such or

(2) Replace such an implementation with its functional specification. These two versions shall be verified to be consistent either by simulating them against each other, by formal verification or by a combination of the two. Once the results are identical, the first cell can be replaced by its functional equivalent.

The order of these two steps can be reversed for the top-down approach. In the top-down approach, instead of an implementation of lower level functions being abstracted to be a higher level function, a higher level function is implemented by some lower level functions. Such refinement of design can be carried out until the implementation at the bottom is completed.

What we have shown is a method for constructing systems from switch-level circuits to functional blocks through successive semantic abstractions. Stated the other way around, we have started with functional specification at the top level and successively refined the specification until it is implemented in the bottom level representation of the medium (in this case, switch-level model of transistors and nodes). In contrast to the conventional view of fixed hierarchical levels, the partitioning of a semantic hierarchy is flexible and problem oriented. Designers can partition each system in the way that is most natural to the design, rather than trying to fit it into rigid pre-defined levels which are not necessarily appropriate.

4. Implementation of the Simulator

The example shown above is very specific, and one can obviously write a multi-level simulator for this particular system. Such a simulator would be of limited use since it only handles designs partitioned in the same way. A general purpose simulator which must support arbitrary levels and mixed-level seems at first unrealistic. We approach this problem by:

1. Separating out the part that is universal to all system levels

2. Using the power of an embedded language [24].

Embedded languages. It has been observed in integrated circuit layout languages that an embedded language — a language supporting

graphics primitives in an existing programming language — has the generality and flexibility in the specification of designs that an interactive graphic layout system usually lacks. The effort of making a graphic system as powerful as an embedded language is essentially that of supporting a general purpose programming language. It is much more sensible to let the compiler of an existing language do the work. The same philosophy applies to a specification language for simulation. We build into a programming language the simulation algorithm and an interactive user interface (corresponding to the debugger in a programming environment) for testing the design. One specifies cells in an embedded simulation language by invoking primitives for transistors, nodes, syntactic cells and semantic cells. These primitives are pre-defined in the language. With the power of a general programming language, users can then specify functional abstractions, the data abstractions, and various data types at any level according to their conceptualization of the design.

Representation of Cells. Cells are represented as *modules* in a programming language supporting separately compilable modules. The language we have used for this specific implementation is Mainsail*[45]. A cell has in it the specification of its constituents, i.e., the implementation in terms of lower level cells, and the procedure for computing its result. The former is supplied by the user and the latter is incorporated automatically. Cells of different natures — primitive, composite, semantic or syntactic — have different ways of computing their result. Each of them is made into a template and the template is compiled together with the specification of the cell's construction in the case of a composition cell. A module is typed according to the template incorporated into it. The net effect is that each module contains the functions necessary to compute its own behavior.

We use a module instead of a procedure to represent each cell because

*Mainsail is a registered trademark of Xidak Inc.

a cell can be mapped directly into a module which has a data section to represent the cell's inputs, outputs and states, and a procedure section for the specification of its constituents or the computation of its behavior. Each module also has one bit of state which records whether any inputs have changed since it reached its last steady state. No computation is necessary if they are the same since the outputs of the last computation are still valid. This "change bit" allows a truly efficient implementation.

Representation of Connections. The computational aspects of a system have attracted much more attention than the communication aspects. Since the transfer of information does cost energy and resources (wires), it must be taken into account. The communication among modules is represented also as a module which contains the connectivity information and various ways to transfer information from an output port to the corresponding input port. Both uni-directional and bi-directional connections can be represented. More important, the connection modules make it possible to simulate a collection of concurrent processes by a sequential machine in an order independent way.

The Universal Fixed-Point Algorithm. By viewing a computational system as an ensemble of cells and connections, we devise a fixed-point algorithm to find the steady state of a cell. The fixed-point algorithm performs a task similar to the so-called "relaxation" algorithm. It can be viewed as an interpreter for a space-time algorithm which describes the system. It takes the implementation of a cell in terms of connections and cells in a bipartite format. The algorithm starts by evaluating all the connections (interpreting ⟨CTERM⟩'s). Since at this point, all data streams are totally undefined functions, only the initially assigned state and input/output values of variables $X_1, X_2, \ldots, X_m$ result. Then each cell is invoked (interpreting ⟨PTERM⟩);

and the state transition function of the cell is evaluated. After all have been invoked, each connection is in turn invoked, to transfer the data and thereby bring about the interactions among connected cells (interpreting (CTERM)'s). This procedure is iterated until the steady state is reached. The bipartite arrangement of cells and connections results in the property that the order in which each cell is invoked is immaterial in the algorithm. Each iteration of the algorithm therefore generates values corresponding to the values of the least fixed-point of the space-time algorithm at some points in the space-time domain. Since the algorithm is to terminate once the steady state is reached, the algorithm yields the correct value which is the corresponding value of the least fixed-point at steady state. Since all cells and connections are represented uniformly at all semantic levels, only a single universal fixed-point algorithm is necessary.

Elements of a Multi-level Simulator. The following modules, embedded into Mainsail, serve as templates for user defined circuits:

1. Cells of various types: transistors, syntactic cells, conductance networks, clocked cells, and functional cells.

2. Connections of various types: nodes, bi-directional connections for syntactic cells below clocked cell level, and uni-directional connections for functional cells. The formation of conductance network topology is also represented as a connection module.

In the object-oriented view of computation such as Simula [2], Smalltalk [18], etc., the above templates are the superclasses of the user defined classes (cells). These templates are the only structure we build into the simulator. Instead of building and maintaining data structures that represent a design, the structure is embedded in user's specification of inter-connected modules.

Hence, no global simulation algorithm is necessary to traverse the data structure.

Each of the modules has variables, constants, and procedures for computing its behavior. Nodes, transistors, conductance networks, the formation of conductance network topology, and clocked cells (transistor networks) are described in Chapter 4 where the behaviors are defined by state transition functions. On the other hand, the behavior of primitive functional cells are defined in a cell library or supplied by the user. Transistors, nodes and primitive functional cells compute directly. A composite cell contains the following lists: sub-modules for the constituents, connections, external input ports and external output ports. It causes its constituents to compute in a recursive manner until one of the primitive functions or abstracted functions is encountered. In the case of a semantic cell, it iterates until reaching its steady state and returns its result. For a syntactic cell, it only computes one iteration and then return its result. The behavior of a bi-directional connection is similar to a node. The behavior of a uni-directional connection is simply transferring data from an output port of one cell to the input port (or ports for broadcasting) of another.

Programming consists of three stages: specifying, compiling, and executing along with debugging of a program; simulation likewise consists of specifying, constructing the structure of a design (compiling), and exercising (executing and debugging) a design.

Specifying a Design. The user interface for specifying a design contains the following keywords and procedures:

(i) *BeginSCell* for a syntactic cell, *BeginCCell* for a composite semantic cell and *BeginFCell* for a abstracted functional cell. These keywords

cause a template containing the procedure for different ways of computing to be created during the "compile time". It also creates the procedure head *Compose* for the user specified constituents. The body of the procedure *Compose* consists of invocations of the following procedures for a user to specify a design.

During "compile time", these invocations cause the template created by the keywords above to be filled with actual instances of the cell's constituents.

(ii) *Node*(*type*); where *type* indicates whether the node is a input node or a internal node.

(iii) *Tran*(*conductanceStrength*, *type*); where *type* indicates enhancement, depletion mode or p-type, n-type transistors.

(iv) *SubModule*(*class*); where *class* indicates a module of certain type, say, an exclusive-or gate.

(v) *Identify*(*fromPort*, *toPort*); where *fromPort* and *toPort* are either nodes in a transistor level design or an input port and an output port for a higher level design. A connection module will be created and attached to the connection list in the cell's template.

(vi) *ExOut*(*port*); and *ExIn*(*port*); for declaring the interface of a cell to the environment. These ports are attached to the corresponding lists in the template.

Compiling. Each composite cell module contains the procedure *Compose* for constructing the internal structure of a design. When this procedure of the top-level module of a design is invoked, it in turn invokes the user specified procedures *SubModule*, *Identify*, *ExIn* or *ExOut*. Procedure

SubModule creates instances of the submodules, updates the submodule list and in turn invokes the *Compose* procedure of the submodules. This action proceeds recursively until a primitive cell specified by, for example, *Node* or *Tran*, or an abstracted functional cell, is reached. A connection between submodules is established when the procedure *Identify* is invoked.

Executing and Debugging. After a network has been constructed, inputs to the network are provided by the user. The simulation of the network starts when the procedure *Compute* of the top-level cell module is invoked. Modules of different nature (primitive or composite, syntactic or semantic) compute differently as described in the previous section.

The debugging of a network is no different than debugging a program and there is no reason not to take advantage of the debugging environment of the programming language in which the simulator is embedded. Inserting a breakpoint into a program text specifies the desired time for a user to examine or force values onto some variables in the program. Similarly, specifying the level of the semantic hierarchy determines the desired time to examine and force values onto ports of modules in a network.

5. Summary

A multi-level simulator which allows user-defined levels instead of rigidly pre-defined levels is described. A clear distinction is drawn between the modularization for ease of specification and for semantic abstraction — the syntactic hierarchy and the semantic hierarchy, respectively. An example of multi-level simulation is given which spans from circuit-level up to the abstract function of an inner product element.

With a formal model as a basis, the implementation of the simulator is simple and uniform at all levels. A single universal fixed-point algorithm is

used. This approach raises the activity of simulation from a low level corresponding to macro assembly level in a programming language to a hierarchical specification corresponding directly to the conceptualization of user's design. Functional abstraction and data abstraction (interfaces between two different levels) of systems have been illustrated. These abstractions are the key to the consistency and efficiency of a multi-level simulator. In simulation, showing that the specification and the implementation are equivalent is not merely desirable but absolutely essential. This working example has shown that formal semantics is an essential feature of any design tool as well as any concurrent programming language.

Chapter 6

Conclusions

1. Summary of the Thesis

A formal methodology for describing concurrent systems has been given. It is based on a model of computation where a system is an ensemble of processes. The language CRYSTAL used in describing the systems has fixed-point semantics. The hierarchical design method now has a firm basis, where the semantics of a space-time algorithm describing an implementation is used as an elementary building block at the next level.

This methodology has been applied to designs spanning several different levels. It is proven to be effective in handling realistic systems. The language can be viewed either as a hardware description language which serves as specification for simulation, or as a programming language for concurrent systems where descriptions can be formally manipulated and the correctness of systems can be proved. The power of this framework lies in providing functional abstraction (fixed-point semantics) and data abstraction (moving from more detailed data types to higher level data types) for system designs. This method of abstraction is essential for meeting the challenge of more and more complex systems.

This methodology, when applied to the simulation of VLSI circuits, results in a simple yet efficient and universal simulator for designs at all

levels. The simulation algorithm is simply the interpretation of the definition of fixed-point. Difficult issues in the area of multi-level and mixed-level simulation have been clarified.

2. Extension to Nondeterministic Systems

In the model presented in Chapter 2, if the control state register is replaced by an oracle which nondeterministically decides the set of input ports to choose and operations described in (viii) and (ix) are replaced by the following, the model becomes a non-deterministic one.

(viii)′ The oracle chooses a set of input ports.

(ix)′ If all of the chosen input ports are filled, we say that the communication is established. The process now starts an operation, called the t'th *invocation* of the process where t is the time coordinate of the process. If some input ports are not filled, the set of chosen input ports are discarded. The process goes back to step (viii)′.

In the deterministic system, the relationship of the time coordinates between neighboring processes is completely determined by the control state generated by a state transition function which in turn is deterministic. Hence the functions that define this relationship are unique and only one space-time algorithm results. In the case of the non-deterministic model, depending on the decision made by the oracle and when various inputs ports are filled, a set of functions which describe the relationships will result. For each of the relationships, there corresponds a space-time algorithm. The semantics of each of the algorithms is obtained the same way as that for a deterministic system; the semantics of the whole system will then be the set of all the individual ones. If the system is determinate, i.e., all the different algorithms

produce the same result, then the behavior of the system can still be described by a state transition function. Otherwise, when it is used in the next level, there are a set of functions corresponding to the system, and the number of algorithms describing the composition of these sets of functions grows exponentially with the number of levels. From a verification point of view, keeping track of the exponential number of possible outcomes is extremely impractical. Statistical treatments of non-deterministic systems in which probabilistic assertions can be made about these systems is a promising approach. This area certainly deserves much more research.

3. Future Work

Silicon Compilation. Thus far, a space-time algorithm has only been used in the behavioral aspect of VLSI systems, namely, formal verification and simulation. To obtain a physical implementation, a general method of obtaining layouts from given sets of layouts of sub-systems must be given. The structural aspect (physical layout) is today approached separately from the behavioral aspect. A system is usually described by two sets of specifications: one for layout, one for simulation. The functionality is verified by simulation and the layout is verified by extracting circuit features (e.g. transistors) and simulating the extracted version. The ideal would be to use one specification that is powerful enough to generate both simulation (interpretation) and layouts (compilation). Our methodology is readily applicable to such an integrated behavioral and structural design tool. Difficulty has been experienced [36] in using a behavioral description of cells consisting of only a few interconnected transistors, and generating the topological information automatically for placing transistor features from this description. Hence at the Clocked Cell level, or syntactic levels below it, a representation [46] of the topological information and feature sizes provided by the user is used and

the behavioral information is extracted from this representation. From these levels up, all specifications will be behavioral. General composition, routing and placement algorithm will generate the structual information automatically. This approach has the advantage that only one specification is ever used: the structural representation at the bottom level and the behavioral representation at the levels above. Once the silicon compiler itself has been verified as correct, each design only need be verified by simulation and formal verification and no consistency check between layout and simulation is necessary.

Automatic Verification of VLSI Systems. Verification of the behavior of systems has been done mainly by simulation. As mentioned earlier in this thesis, it is practically impossible to verify the behavior completely by simulation. The space-time algorithm is suitable for automatic verification since it is based on the typed λ-calculus [13]. The use of an appropriate symbolic manipulation system for verifying large systems is a favored approach for dealing with the increasing complexity of VLSI. Such a technique could, in principle, replace simulation for verification at every level even down to the transistors.

Analysis and Synthesis Techniques. Space-time algorithms provide a way of describing concurrent systems; the fixed-point semantics and induction principles allow verification of these systems. What needs to be developed is a calculus for manipulating the space-time equations so that systems can be synthesized from their specification. In linear systems, matrix theory, z-transform notation and its calculus provide powerful tools for manipulating the description of these systems. Numerous synthesis techniques have evolved from these analysis techniques. Techniques for more general classes of applications will facilitate both silicon compilation (synthesis) and automatic

verification (analysis) of these systems.

References

[1] Backus, J. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,* CACM, 21(8)613-641, August 1978.

[2] Birtwhistle, G. M., Dahl, O-J, Myhrhaug, B., and Nygaard, K., *Simula Begin,* Petrocelli, New York, 1973.

[3] Bryant, R.E., *A Switch-level Simulation Model for Integrated Logic Circuits,* Doctoral Dissertation, Massachusetts Institute of Technology, Cambridge Massachusetts, March, 1981.

[4] Bryant, R., Schuster, M., Whiting D. *MOSSIM II: A Switch-level Simulation for MOS LSI, User's Manual* Technical Report 5033:TR:82, California Institute of Technology, 1982.

[5] Burstall, R.M., *Proving Properties of Programs by Structural Induction,* Computing Journal, 12(1): 41-48, February, 1969.

[6] Carnap, R., *Introduction to Symbolic Logic and its Applications,* Dover Publications, Inc. New York, 1958.

[7] Cardelli, L., *An Algebraic Approach to hardware Description and Verification,* Doctoral Dissertation, Department of Computer Science, University of Edinburgh, 1982.

[8] Chen, M.C. and Mead C.A., *Concurrent Algorithms as Space-time Recursion Equations,* Proceedings of USC Workshop on VLSI and Modern Signal Processing, pages 31-52, Los Angles, November, 1982.

[9] Chen, M.C. and Mead C.A., *A Hierarchical Simulator Based on Formal Semantics,* Proceedings of the Third Caltech Conference on VLSI, pages 207-223, Pasadena, March, 1983.

[10] Clinger, W.D., *Foundations of Actor Semantics,* Doctoral Dissertation, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1981.

[11] Dijkstra, E. W., *Cooperating sequential processes,* In F.Genuys, editor, Programming Languages, pages 43–112. Academic Press, New York, 1968.

[12] Gardner, R.I., Grundmann, J.W., Pass, D.S., and Swanson, K.C., *Multilevel Simulation in VISTA: Architecture and Performance,* Proceedings of ICCC 82, New York, 1982.

[13] Gordon, M., *A Very Simple Model of Sequential Behaviour of NMOs,* Proceedings of VLSI 81, Edinburgh, Academic Press, 1981.

[14] Gordon, M.J., Milner, A.J., and Wadsworth, C.P., *Edinburgh LCF,* Lecture Notes in Computer Science, Springer-Verlag, New York, 1979.

[15] Hewitt, C., and Baker, H., *Laws for Communicating parallel Processes,* Proceedings of IFIP Congress 77, Toronto, 1977, pages 987-992.

[16] Hoare, C.A.R., *Communicating Sequential Processes,* CACM, 21(8)666-677, August 1978.

[17] Horning, J.J., and Randell, B., *Process Structuring,* Computing Surveys, 5(1)5-30, March 1973.

[18] Ingalls, D., *The Smalltalk 76 Programming System: Design and Implementation,* Proceedings of the Fifth ACM Conference on Principles of Programming Systems, pages 9-16, January 1978.

[19] Johnsson, L., Weiser, U., Cohen, D. and Davis, A., *Towards a Formal Treatment of VLSI Arrays,* Proceedings of the Second Caltech Conference on VLSI, pages 375-398, Pasadena, January, 1981.

[20] Kahn, G., Proceedings of IFIP Congress 74, *The Semantics of a Simple Language for Parallel Programming*, 1974.

[21] Kung, H.T., *Why Systolic Architectures?*, IEEE Computer, pages 37-46, January 1982.

[22] Kung, H.T. and Leiserson C.E., *Algorithms for VLSI Processor Arrays*, in C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980, chapter 8.3.

[23] Lin T.Z. and Mead C.A., *The Application of Group Theory in Classifying Systolic Arrays*, Display File #5006, California Institute of Technology, March, 1982.

[24] Locanthi, B., *LAP: A Simula Package for IC Layout*, Caltech SSP Report #1862, California Institute of Technology, 1978

[25] Lyon R. F., Two's Complement Pipeline Multipliers, *IEEE Trans. on Communications* COM-24, pp. 418-425, 1976.

[26] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[27] Mead C. and Conway L. *Introduction to VLSI Systems* Addison-Wesley, New York 1980.

[28] Milne, G. J., *CIRCAL, A Calculus for Circuit Description*, CSR-122-82, Department of Computer Science, University of Edinburgh, 1982.

[29] Milner, R., *Models of LCF*, AIM-186/CS-332, Computer Science Department, Stanford University, 1973.

[30] Milner, R., *A Calculus of Communicating Systems,* Lecture Notes in Computer Science 92, Springer-Verlag, New York, 1980.

[31] Nurie, G.M., *LOGAL+ — A Hardware Description Language for Hierarchical Design and Multilevel Simulation,* Proceedings of ICCC 82, New York, 1982.

[32] Peterson, J.L., *Petri-net Theory and the Modeling of Systems,* Prentice-Hall, Englewood Cliffs, 1981.

[33] Petri, C.A., *Kommunikation mit Automaten,* Doctoral Disertation, University of Bonn, West Germany, 1962, (in German); also MIT Memorandum MAC-M-212, Project MAC, MIT, Cambridge, Mass.

[34] Rem, M., van de Snepscheut, J.L.A., and Udding, J.T., *Trace Theory and the Definition of Hierarchical Components,* Proceedings of the Third Caltech Conference on VLSI, pages 225-239, Pasadena, March, 1983.

[35] Rowson, J. A., *Understanding Hierarchical Design,* Doctoral Dissertation, California Institute of Technology, April, 1980.

[36] Rupp, C.R., *Components of A Silicon Compiler System,* Proceedings of VLSI 81, Edinburgh, Academic Press, 1981.

[37] Scott, D., *Outline of a Mathematical Theory of Computation,* Oxford Mon. PRG-2, Oxford University Press, Oxford,England, 1970.

[38] Scott, D. and Strachey, C., *Toward a Mathematical Semantics for Computer Languages* Fox, J., editor. Polytechnic institute of Brooklyn Press, New York, 1971.

[39] Seitz, C., *System Timing,* in C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980, chapter 7.

[40] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach,* MIT Press, Cambridge, Massachusetts, 1977.

[41] Sutherland I. and Mead C., *Micro-electronics and Computer Science,* Scientific American 237(3):210-229, September, 1977.

[42] Von Neumann, J. *Theory of Self-Reproducing Automata,* (edited and Completed by Burks, A. W.), University of Illinois Press, Urbana and London, 1966

[43] Vuillemin, J. E., *Proof Techniques for Recursive Programs,* Doctoral Dissertation, Computer Science Department, Stanford University, 1974.

[44] Wawrzynek J. and Lin T. M., *A bit Serial Architecture for Multiplication and Interpolation,* 5067:DF:83, Computer Science Department, California Institute of Technology, January, 1983.

[45] Wilcox, C.R., Dageforde, M.L., and Jirak, G.A., *Mainsail(TM) Language Manual VErsion 4.0,* Xidak, 1979.

[46] Witney,T. and Mead, C., *Pooh: A Uniform Representation For Circuit Level Designs,* To appear in Proceedings of VLSI 83, Trondheim, 1983.